

numpy (version 1.11.0)[index](#)
/usr/lib/python2.7/dist-packages/numpy/_init_.pyNumPy
=====Provides
1. An array [object](#) of arbitrary homogeneous items
2. Fast mathematical operations over arrays
3. Linear Algebra, Fourier Transforms, Random Number Generation

How to use the documentation

Documentation is available in two forms: docstrings provided with the code, and a loose standing reference guide, available from `the NumPy homepage <<http://www.scipy.org>>`_.We recommend exploring the docstrings using `IPython <<http://ipython.scipy.org>>`_, an advanced Python shell with TAB-completion and introspection capabilities. See below for further instructions.

The docstring examples assume that `numpy` has been imported as `np`::

>>> import numpy as np

Code snippets are indicated by three greater-than signs::

>>> x = 42
>>> x = x + 1

Use the built-in ``help`` function to view a function's docstring::

>>> help(np.sort)
... # doctest: +SKIPFor some objects, ``np.[info](#)(obj)`` may provide additional help. This is particularly true if you see the line "Help on [ufunc object](#):" at the top of the help() page. Ufuncs are implemented in C, not Python, for speed. The native Python help() does not know how to view their help, but our np.[info\(\)](#) function does.

To search for documents containing a keyword, do::

>>> np.[lookfor](#)('keyword')
... # doctest: +SKIP

General-purpose documents like a glossary and help on the basic concepts of numpy are available under the ``doc`` sub-module::

>>> from numpy import doc
>>> help(doc)
... # doctest: +SKIP

Available subpackages

doc Topical documentation on broadcasting, indexing, etc.

lib Basic functions used by several sub-packages.

random Core Random Tools

linalg Core Linear Algebra Tools

fft Core FFT routines

polynomial Polynomial tools

testing Numpy testing tools

f2py Fortran to Python Interface Generator.

distutils Enhancements to distutils with support for

Fortran compilers support and more.

Utilities

test Run numpy unittests

show_config Show numpy build configuration

dual Overwrite certain functions with high-performance Scipy tools

matlib Make everything matrices.

```

__version__
    NumPy version string

Viewing documentation using IPython
-----
Start IPython with the NumPy profile ('`ipython -p numpy``), which will
import `numpy` under the alias `np`. Then, use the ``cpaste`` command to
paste examples into the shell. To see which functions are available in
`numpy`, type ``np.<TAB>`` (where ``<TAB>`` refers to the TAB key), or use
``np.*cos?<ENTER>`` (where ``<ENTER>`` refers to the ENTER key) to narrow
down the list. To view the docstring for a function, use
``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to view
the source code).

```

Copies vs. in-place operation

Most of the functions in `numpy` return a copy of the array argument
(e.g., `np.sort`). In-place versions of these functions are often
available as array methods, i.e. ``x = np.array([1,2,3]); x.sort()``.
Exceptions to this rule are documented.

Package Contents

config	ctypeslib	lib (package)	polynomial (package)
import_tools	distutils (package)	linalg (package)	random (package)
add_newdocs	dual	ma (package)	setup
compat (package)	f2py (package)	matlib	testing (package)
core (package)	fft (package)	matrixlib (package)	version

Classes

builtin_.object
broadcast
busdaycalendar
dtype
flatiter
generic
bool
datetime64
flexible
character
 string_(_builtin_.str, character)
 unicode_(_builtin_.unicode, character)
void
 record
number
 inexact
 complexfloating
 complex128(complexfloating, _builtin_.complex)
 complex256
 complex64
 floating
 float128
 float16
 float32
 float64(floating, _builtin_.float)
 integer
 signedinteger
 int16
 int32
 int64(signedinteger, _builtin_.int)
 int64(signedinteger, _builtin_.int)
 int8
 timedelta64
 unsignedinteger
 uint16
 uint32
 uint64

```
    uint64
    uint8
    object_
ndarray
recarray
numpy.core.defchararray.chararray
numpy.core.memmap.memmap
numpy.matrixlib.defmatrix.matrix
nditer
ufunc
numpy_. import_tools.PackageLoader
numpy.core.getlimits.finfo
numpy.core.getlimits.iinfo
numpy.core.machar.MachAr
numpy.core.numeric.errstate
numpy.lib._datasource.DataSource
numpy.lib.function_base.vectorize
numpy.lib.index_tricks.ndenumerate
numpy.lib.index_tricks.ndindex
numpy.lib.polynomial.poly1d
_builtin_.str(_builtin_.basestring)
_string_(_builtin_.str, character)
_builtin_.unicode(_builtin_.basestring)
_unicode_(_builtin_.unicode, character)
exceptions.DeprecationWarning(exceptions.Warning)
    ModuleDeprecationWarning
exceptions.RuntimeError(exceptions.StandardError)
    numpy.core. internal.TooHardError
exceptions.RuntimeWarning(exceptions.Warning)
    numpy.core.numeric.ComplexWarning
exceptions.UserWarning(exceptions.Warning)
    VisibleDeprecationWarning
    numpy.lib.polynomial.RankWarning
numpy.core.records.format_parser
```

class ComplexWarning(exceptions.RuntimeWarning)

The warning raised when casting a [complex dtype](#) to a real [dtype](#).

As implemented, casting a [complex number](#) to a real discards its imaginary part, but this behavior may not be what the user actually wants.

Method resolution order:

```
ComplexWarning
exceptions.RuntimeWarning
exceptions.Warning
exceptions.Exception
exceptions.BaseException
    builtin_.object
```

Data descriptors defined here:

weakref

list of weak references to the object (if defined)

Methods inherited from [exceptions.RuntimeWarning](#):

init(...)

x. [_init_](#)(...) initializes x; see help(type(x)) for signature

Data and other attributes inherited from [exceptions.RuntimeWarning](#):

new = <built-in method __new__ of type object>

T. [_new_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

Methods inherited from [exceptions.BaseException](#):

```
_delattr__(...)
    x.\_delattr\_\_('name') <==> del x.name

_getattribute__(...)
    x.\_getattribute\_\_('name') <==> x.name

_getitem__(...)
    x.\_getitem\_\_(y) <==> x[y]

_getslice__(...)
    x.\_getslice\_\_(i, j) <==> x[i:j]

    Use of negative indices is not supported.

_reduce__(...)

_repr__(...)
    x.\_repr\_\_() <==> repr(x)

_setattr__(...)
    x.\_setattr\_\_('name', value) <==> x.name = value

_setstate__(...)

_str__(...)
    x.\_str\_\_() <==> str(x)

_unicode__(...)
```

Data descriptors inherited from [exceptions.BaseException](#):

```
_dict_
args
message
```

```
class DataSource(\_builtin\_.object)
    DataSource(destpath='.')
        A generic data source file (file, http, ftp, ...).

    DataSources can be local files or remote files/URLs. The files may
    also be compressed or uncompressed. DataSource hides some of the
    low-level details of downloading the file, allowing you to simply pass
    in a valid file path (or URL) and obtain a file object.

    Parameters
    -----
    destpath : str or None, optional
        Path to the directory where the source file gets downloaded to for
        use. If `destpath` is None, a temporary directory will be created.
        The default path is the current directory.

    Notes
    -----
    URLs require a scheme string ('`http://`') to be used, without it they
    will fail::

        >>> repos = DataSource()
        >>> repos.exists('www.google.com/index.html')
        False
        >>> repos.exists('http://www.google.com/index.html')
        True

    Temporary directories are deleted when the DataSource is deleted.

    Examples
    -----
    ::

        >>> ds = DataSource('/home/guido')
        >>> urlname = 'http://www.google.com/index.html'
        >>> gfile = ds.open('http://www.google.com/index.html') # remote file
        >>> ds.abspath(urlname)
        '/home/guido/www.google.com/site/index.html'
```

```
>>> ds = DataSource(None) # use with temporary file
>>> ds.open('/home/guido/foobar.txt')
<open file '/home/guido.foobar.txt', mode 'r' at 0x91d4430>
>>> ds.abspath('/home/guido/foobar.txt')
'/tmp/tmpy4pgsP/home/guido/foobar.txt'

Methods defined here:

__del__(self)

__init__(self, destpath='.')
    Create a DataSource with a local path at destpath.

abspath(self, path)
    Return absolute path of file in the DataSource directory.

    If `path` is an URL, then `abspath` will return either the location
    the file exists locally or the location it would exist when opened
    using the `open` method.

    Parameters
    -----
    path : str
        Can be a local file or a remote URL.

    Returns
    -----
    out : str
        Complete path, including the DataSource' destination directory.

    Notes
    -----
    The functionality is based on `os.path.abspath`.

exists(self, path)
    Test if path exists.

    Test if `path` exists as (and in this order):
    - a local file.
    - a remote URL that has been downloaded and stored locally in the
      DataSource directory.
    - a remote URL that has not been downloaded, but is valid and
      accessible.

    Parameters
    -----
    path : str
        Can be a local file or a remote URL.

    Returns
    -----
    out : bool
        True if `path` exists.

    Notes
    -----
    When `path` is an URL, `exists` will return True if it's either
    stored locally in the DataSource directory, or is a valid remote
    URL. DataSource does not discriminate between the two, the file
    is accessible if it exists in either location.

open(self, path, mode='r')
    Open and return file-like object.

    If `path` is an URL, it will be downloaded, stored in the
    DataSource directory and opened from there.

    Parameters
    -----
    path : str
        Local file path or URL to open.
    mode : {'r', 'w', 'a'}, optional
        Mode to open `path`. Mode 'r' for reading, 'w' for writing,
        'a' to append. Available modes depend on the type of object
        specified by `path`. Default is 'r'.

    Returns
    -----
    out : file object
        File object.
```

Data descriptors defined here:

```

__dict__
    dictionary for instance variables (if defined)

__weakref__
    list of weak references to the object (if defined)

class MachAr(builtin object)
    Diagnosing machine parameters.

    Attributes
    -----
    ibeta : int
        Radix in which numbers are represented.
    it : int
        Number of base-`ibeta` digits in the floating point mantissa M.
    machep : int
        Exponent of the smallest (most negative) power of `ibeta` that,
        added to 1.0, gives something different from 1.0.
    eps : float
        Floating-point number ``beta**machep`` (floating point precision)
    negep : int
        Exponent of the smallest power of `ibeta` that, subtracted
        from 1.0, gives something different from 1.0.
    epsneg : float
        Floating-point number ``beta**negep``.
    iexp : int
        Number of bits in the exponent (including its sign and bias).
    minexp : int
        Smallest (most negative) power of `ibeta` consistent with there
        being no leading zeros in the mantissa.
    xmin : float
        Floating point number ``beta**minexp`` (the smallest [in
        magnitude] usable floating value).
    maxexp : int
        Smallest (positive) power of `ibeta` that causes overflow.
    xmax : float
        `` $(1 - \text{epsneg}) * \text{beta}^{\text{maxexp}}$ `` (the largest [in magnitude]
        usable floating value).
    irnd : int
        In ``range(6)``, information on what kind of rounding is done
        in addition, and on how underflow is handled.
    ngrd : int
        Number of 'guard digits' used when truncating the product
        of two mantissas to fit the representation.
    epsilon : float
        Same as `eps`.
    tiny : float
        Same as `xmin`.
    huge : float
        Same as `xmax`.
    precision : float
        `` $\text{int}(-\log_{10}(\text{eps}))$ ``.
    resolution : float
        `` $10^{(-\text{precision})}$ ``.

    Parameters
    -----
    float_conv : function, optional
        Function that converts an integer or integer array to a float
        or float array. Default is float.
    int_conv : function, optional
        Function that converts a float or float array to an integer or
        integer array. Default is int.
    float_to_float : function, optional
        Function that converts a float array to float. Default is float.
        Note that this does not seem to do anything useful in the current
        implementation.
    float_to_str : function, optional
        Function that converts a single float to a string. Default is
        ``lambda v: '%24.16e' % v``.
    title : str, optional
        Title that is printed in the string representation of `MachAr`.

    See Also
    -----
    finfo : Machine limits for floating point types.
    iinfo : Machine limits for integer types.

    References
    -----
    .. [1] Press, Teukolsky, Vetterling and Flannery,
        "Numerical Recipes in C++," 2nd ed.,
        Cambridge University Press, 2002, p. 31.

```

Methods defined here:

```
__init__(self, float_conv=<type 'float'>, int_conv=<type 'int'>, float_to_float=<type 'float'>, float_to_str=<function <lambd>>, title='Python floating point number')
    float_conv - convert integer to float (array)
    int_conv   - convert float (array) to integer
    float_to_float - convert float array to float
    float_to_str - convert array float to str
    title      - description of used floating point numbers

__str__(self)
```

Data descriptors defined here:

__dict__	dictionary for instance variables (if defined)
__weakref__	list of weak references to the object (if defined)

class [ModuleDeprecationWarning](#)([exceptions.DeprecationWarning](#))

Module deprecation warning.

The nose tester turns ordinary Deprecation warnings into test failures. That makes it hard to deprecate whole modules, because they get imported by default. So this is a special Deprecation warning that the nose tester will let pass without making tests fail.

Method resolution order:

```
ModuleDeprecationWarning
exceptions.DeprecationWarning
exceptions.Warning
exceptions.Exception
exceptions.BaseException
builtin\_object
```

Data descriptors defined here:

__weakref__	list of weak references to the object (if defined)
--------------------	--

Methods inherited from [exceptions.DeprecationWarning](#):

__init__ (...)	x. __init__ (...) initializes x; see help(type(x)) for signature
-----------------------	--

Data and other attributes inherited from [exceptions.DeprecationWarning](#):

__new__ = <built-in method __new__ of type object>	T. __new__ (S, ...) -> a new object with type S, a subtype of T
---	---

Methods inherited from [exceptions.BaseException](#):

__delattr__ (...)	x. __delattr__ ('name') <==> del x.name
--------------------------	---

__getattribute__ (...)	x. __getattribute__ ('name') <==> x.name
-------------------------------	--

__getitem__ (...)	x. __getitem__ (y) <==> x[y]
--------------------------	--

__getslice__ (...)	x. __getslice__ (i, j) <==> x[i:j]
---------------------------	--

Use of negative indices is not supported.

__reduce__ (...)	
-------------------------	--

__repr__ (...)	
-----------------------	--

```

x.__repr__() <==> repr(x)

setattr(...)
    x.setattr('name', value) <==> x.name = value

setstate(...)

str(...)
    x.str() <==> str(x)

unicode(...)

```

Data descriptors inherited from [exceptions.BaseException](#):

dict

args

message

class **PackageLoader**([builtin.object](#))

Methods defined here:

call(self, *packages, **options)

Load one or more packages into parent package top-level namespace.

This function is intended to shorten the need to import many subpackages, say of scipy, constantly with statements such as

```
import scipy.linalg, scipy.fftpack, scipy/etc...
```

Instead, you can say:

```
import scipy
scipy.pkgload('linalg','fftpack',...)
```

or

```
scipy.pkgload()
```

to load all of them in one call.

If a name which doesn't exist in scipy's namespace is given, a warning is shown.

Parameters

***packages** : arg-tuple
 the names (one or more strings) of all the modules one
 wishes to load into the top-level namespace.
verbose= : [integer](#)
 verbosity level [default: -1].
 verbose=-1 will suspend also warnings.
force= : [bool](#)
 when True, force reloading loaded packages [default: False].
postpone= : [bool](#)
 when True, don't load packages [default: False]

init(self, verbose=False, infunc=False)

Manages loading packages.

error(self, mess)

get_pkgdocs(self)

Return documentation summary of subpackages.

log(self, mess)

warn(self, mess)

Data descriptors defined here:

dict

dictionary for instance variables (if defined)

weakref

list of weak references to the object (if defined)

class **RankWarning**([exceptions.UserWarning](#))
Issued by `polyfit` when the Vandermonde [matrix](#) is rank deficient.
For more information, a way to suppress the warning, and an example of [RankWarning](#) being issued, see `polyfit`.

Method resolution order:

[RankWarning](#)
[exceptions.UserWarning](#)
[exceptions.Warning](#)
[exceptions.Exception](#)
[exceptions.BaseException](#)
[builtin.object](#)

Data descriptors defined here:

__weakref__
list of weak references to the object (if defined)

Methods inherited from [exceptions.UserWarning](#):

__init__(...)
x. [__init__](#)(...) initializes x; see help(type(x)) for signature

Data and other attributes inherited from [exceptions.UserWarning](#):

__new__ = <built-in method __new__ of type object>
T. [__new__](#)(S, ...) -> a new [object](#) with type S, a subtype of T

Methods inherited from [exceptions.BaseException](#):

__delattr__(...)
x. [__delattr__](#)('name') <=> del x.name
__getattribute__(...)
x. [__getattribute__](#)('name') <=> x.name
__getitem__(...)
x. [__getitem__](#)(y) <=> x[y]
__getslice__(...)
x. [__getslice__](#)(i, j) <=> x[i:j]

Use of negative indices is not supported.

__reduce__(...)
__repr__(...)
x. [__repr__](#)() <=> repr(x)
__setattr__(...)
x. [__setattr__](#)('name', value) <=> x.name = value
__setstate__(...)
__str__(...)
x. [__str__](#)() <=> [str](#)(x)
__unicode__(...)

Data descriptors inherited from [exceptions.BaseException](#):

__dict__
args
message

```
class TooHardError(exceptions.RuntimeError)
    # Exception used in shares memory\(\)
```

Method resolution order:

```
TooHardError
exceptions.RuntimeError
exceptions.StandardError
exceptions.Exception
exceptions.BaseException
builtin\_.object
```

Data descriptors defined here:

[__weakref__](#)

list of weak references to the object (if defined)

Methods inherited from [exceptions.RuntimeError](#):

[__init__\(...\)](#)

x. [__init__](#)(...) initializes x; see help(type(x)) for signature

Data and other attributes inherited from [exceptions.RuntimeError](#):

[__new__](#) = <built-in method __new__ of type object>

T. [__new__](#)(S, ...) -> a new [object](#) with type S, a subtype of T

Methods inherited from [exceptions.BaseException](#):

[__delattr__\(...\)](#)

x. [__delattr__](#)('name') <==> del x.name

[__getattribute__\(...\)](#)

x. [__getattribute__](#)('name') <==> x.name

[__getitem__\(...\)](#)

x. [__getitem__](#)(y) <==> x[y]

[__getslice__\(...\)](#)

x. [__getslice__](#)(i, j) <==> x[i:j]

Use of negative indices is not supported.

[__reduce__\(...\)](#)

[__repr__\(...\)](#)

x. [__repr__](#)() <==> repr(x)

[__setattr__\(...\)](#)

x. [__setattr__](#)('name', value) <==> x.name = value

[__setstate__\(...\)](#)

[__str__\(...\)](#)

x. [__str__](#)() <==> str(x)

[__unicode__\(...\)](#)

Data descriptors inherited from [exceptions.BaseException](#):

[__dict__](#)

[args](#)

[message](#)

```
class VisibleDeprecationWarning(exceptions.UserWarning)
```

Visible deprecation warning.

By default, python will not show deprecation warnings, so this class can be used when a very visible warning is helpful, for example because the usage is most likely a user bug.

Method resolution order:

[VisibleDeprecationWarning](#)
[exceptions.UserWarning](#)
[exceptions.Warning](#)
[exceptions.Exception](#)
[exceptions.BaseException](#)
[builtin._object](#)

Data descriptors defined here:

__weakref__

list of weak references to the object (if defined)

Methods inherited from [exceptions.UserWarning](#):

__init__(...)

x. [__init__](#)(...) initializes x; see help(type(x)) for signature

Data and other attributes inherited from [exceptions.UserWarning](#):

__new__ = <built-in method __new__ of type object>

T. [__new__](#)(S, ...) -> a new [object](#) with type S, a subtype of T

Methods inherited from [exceptions.BaseException](#):

__delattr__(...)

x. [__delattr__](#)('name') <==> del x.name

__getattribute__(...)

x. [__getattribute__](#)('name') <==> x.name

__getitem__(...)

x. [__getitem__](#)(y) <==> x[y]

__getslice__(...)

x. [__getslice__](#)(i, j) <==> x[i:j]

Use of negative indices is not supported.

__reduce__(...)

__repr__(...)

x. [__repr__](#)() <==> repr(x)

__setattr__(...)

x. [__setattr__](#)('name', value) <==> x.name = value

__setstate__(...)

__str__(...)

x. [__str__](#)() <==> [str](#)(x)

__unicode__(...)

Data descriptors inherited from [exceptions.BaseException](#):

__dict__

args

message

bool8 = class [bool_\(generic\)](#)

Numpy's Boolean type. Character code: ``?``. Alias: [bool8](#)

Method resolution order:

[bool](#)
[generic](#)
[builtin_object](#)

Methods defined here:

[**__and__\(...\)**](#)
 $x.\underline{\text{and}}(y) \iff x\&y$

[**__eq__\(...\)**](#)
 $x.\underline{\text{eq}}(y) \iff x==y$

[**__ge__\(...\)**](#)
 $x.\underline{\text{ge}}(y) \iff x>=y$

[**__gt__\(...\)**](#)
 $x.\underline{\text{gt}}(y) \iff x>y$

[**__hash__\(...\)**](#)
 $x.\underline{\text{hash}}() \iff \text{hash}(x)$

[**__index__\(...\)**](#)
 $x[y:z] \iff x[y.\underline{\text{index}}():z.\underline{\text{index}}()]$

[**__le__\(...\)**](#)
 $x.\underline{\text{le}}(y) \iff x<=y$

[**__lt__\(...\)**](#)
 $x.\underline{\text{lt}}(y) \iff x<y$

[**__ne__\(...\)**](#)
 $x.\underline{\text{ne}}(y) \iff x!=y$

[**__nonzero__\(...\)**](#)
 $x.\underline{\text{nonzero}}() \iff x != 0$

[**__or__\(...\)**](#)
 $x.\underline{\text{or}}(y) \iff x|y$

[**__rand__\(...\)**](#)
 $x.\underline{\text{rand}}(y) \iff y\&x$

[**__ror__\(...\)**](#)
 $x.\underline{\text{ror}}(y) \iff y|x$

[**__rxor__\(...\)**](#)
 $x.\underline{\text{rxor}}(y) \iff y^x$

[**__xor__\(...\)**](#)
 $x.\underline{\text{xor}}(y) \iff x^y$

Data and other attributes defined here:

[**__new__** = <built-in method `__new__` of type object>
 \$T.\underline{\text{new}}\(S, \dots\) \rightarrow\$ a new \[object\]\(#\) with type \$S\$, a subtype of \$T\$](#)

Methods inherited from [generic](#):

[**__abs__\(...\)**](#)
 $x.\underline{\text{abs}}() \iff \text{abs}(x)$

[**__add__\(...\)**](#)
 $x.\underline{\text{add}}(y) \iff x+y$

[**__array__\(...\)**](#)
 $\text{sc}.\underline{\text{array}}([\text{type}]) \text{ return } 0\text{-dim array}$

[**__array_wrap__\(...\)**](#)
 $\text{sc}.\underline{\text{array_wrap}}(\text{obj}) \text{ return scalar from array}$

```
__copy__(...)  
__deepcopy__(...)  
__div__(...)  
    x.__div__(y) <==> x/y  
__divmod__(...)  
    x.__divmod__(y) <==> divmod(x, y)  
__float__(...)  
    x.__float__() <==> float(x)  
__floordiv__(...)  
    x.__floordiv__(y) <==> x//y  
__format__(...)  
    NumPy array scalar formatter  
__getitem__(...)  
    x.__getitem__(y) <==> x[y]  
__hex__(...)  
    x.__hex__() <==> hex(x)  
__int__(...)  
    x.__int__() <==> int(x)  
__invert__(...)  
    x.__invert__() <==> ~x  
__long__(...)  
    x.__long__() <==> long(x)  
__lshift__(...)  
    x.__lshift__(y) <==> x<<y  
__mod__(...)  
    x.__mod__(y) <==> x%y  
__mul__(...)  
    x.__mul__(y) <==> x*y  
__neg__(...)  
    x.__neg__() <==> -x  
__oct__(...)  
    x.__oct__() <==> oct(x)  
__pos__(...)  
    x.__pos__() <==> +x  
__pow__(...)  
    x.__pow__(y[, z]) <==> pow(x, y[, z])  
__radd__(...)  
    x.__radd__(y) <==> y+x  
__rdiv__(...)  
    x.__rdiv__(y) <==> y/x  
__rdivmod__(...)  
    x.__rdivmod__(y) <==> divmod(y, x)  
__reduce__(...)  
__repr__(...)  
    x.__repr__() <==> repr(x)  
__rfloordiv__(...)  
    x.__rfloordiv__(y) <==> y//x  
__rlshift__(...)  
    x.__rlshift__(y) <==> y<<x
```

__rmod__(...)
x. rmod(y) <==> y%x

__rmul__(...)
x. rmul(y) <==> y*x

__rpow__(...)
y. rpow(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x. rrshift(y) <==> y>>x

__rshift__(...)
x. rshift(y) <==> x>>y

__rsub__(...)
x. rsub(y) <==> y-x

__rtruediv__(...)
x. rtruediv(y) <==> y/x

__setstate__(...)

__sizeof__(...)

__str__(...)
x. str() <==> str(x)

__sub__(...)
x. sub(y) <==> x-y

__truediv__(...)
x. truediv(y) <==> x/y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters
-----
new_order : str, optional
    Byte order to force; a value from the byte order specifications
    above. The default value ('S') results in swapping the current
    byte order. The code does a case-insensitive check on the first
    letter of `new_order` for the alternatives above. For example,
    any of 'B' or 'b' or 'biggish' are valid to specify big-endian.
```

Returns

new_dtype : dtype
 New `dtype` object with the given change to the [byte](#) order.

nonzero(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

toString(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

Data descriptors inherited from [generic](#):

T
transpose

_array_interface_
Array protocol: Python side

_array_priority_
Array priority.

_array_struct_
Array protocol: struct

base
base object

data
pointer to start of data

dtype
get array data-descriptor

flags
integer value of flags

flat
a 1-d view of scalar

imag
imaginary part of scalar

itemsize
length of one element in bytes

nbytes
length of item in bytes

ndim
number of array dimensions

real
real part of scalar

shape
tuple of array dimensions

size

number of elements in the gentype

strides

tuple of bytes steps in each dimension

class `bool_(generic)`

Numpy's Boolean type. Character code: ``?``. Alias: `bool8`

Method resolution order:

`bool`
`generic`
`builtin_object`

Methods defined here:

`__and__(...)`
`x. __and__(y) <==> x&y`

`__eq__(...)`
`x. __eq__(y) <==> x==y`

`__ge__(...)`
`x. __ge__(y) <==> x>=y`

`__gt__(...)`
`x. __gt__(y) <==> x>y`

`__hash__(...)`
`x. __hash__() <==> hash(x)`

`__index__(...)`
`x[y:z] <==> x[y. __index__():z. __index__()]`

`__le__(...)`
`x. __le__(y) <==> x<=y`

`__lt__(...)`
`x. __lt__(y) <==> x<y`

`__ne__(...)`
`x. __ne__(y) <==> x!=y`

`__nonzero__(...)`
`x. __nonzero__() <==> x != 0`

`__or__(...)`
`x. __or__(y) <==> x|y`

`__rand__(...)`
`x. __rand__(y) <==> y&x`

`__ror__(...)`
`x. __ror__(y) <==> y|x`

`__rxor__(...)`
`x. __rxor__(y) <==> y^x`

`__xor__(...)`
`x. __xor__(y) <==> x^y`

Data and other attributes defined here:

`__new__ = <built-in method __new__ of type object>`
`T. __new__(S, ...) -> a new object with type S, a subtype of T`

Methods inherited from `generic`:

`__abs__(...)`

```
x.abs() <==> abs(x)

add(...)
x.add(y) <==> x+y

array(...)
sc.array(|type) return 0-dim array

array_wrap(...)
sc.array_wrap(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
x.div(y) <==> x/y

divmod(...)
x.divmod(y) <==> divmod(x, y)

float(...)
x.float() <==> float(x)

floordiv(...)
x.floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x.getitem(y) <==> x[y]

hex(...)
x.hex() <==> hex(x)

int(...)
x.int() <==> int(x)

invert(...)
x.invert() <==> ~x

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

neg(...)
x.neg() <==> -x

oct(...)
x.oct() <==> oct(x)

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)
```

__reduce__(...)
Not implemented (virtual attribute)

__repr__(...)
x. __repr__() <==> repr(x)

__rfloordiv__(...)
x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
x. __rlshift__(y) <==> y<<x

__rmod__(...)
x. __rmod__(y) <==> y%x

__rmul__(...)
x. __rmul__(y) <==> y*x

__rpow__(...)
y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x. __rrshift__(y) <==> y>>x

__rshift__(...)
x. __rshift__(y) <==> x>>y

__rsub__(...)
x. __rsub__(y) <==> y-x

__rtruediv__(...)
x. __rtruediv__(y) <==> y/x

__setstate__(...)

__sizeof__(...)

__str__(...)
x. __str__() <==> str(x)

__sub__(...)
x. __sub__(y) <==> x-y

__truediv__(...)
x. __truediv__(y) <==> x/y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:
```

```
* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters
-----
new_order : str, optional
    Byte order to force; a value from the byte order specifications
    above. The default value ('S') results in swapping the current
    byte order. The code does a case-insensitive check on the first
    letter of `new_order` for the alternatives above. For example,
    any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns
-----
new_dtype : dtype
    New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
```

The corresponding attribute of the derived class of interest.

reshape(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.
```

```
trace(...)
    Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

transpose(...)
    Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

var(...)
    Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

view(...)
    Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.
```

Data descriptors inherited from [generic](#):

T
 transpose

_array_interface_
 Array protocol: Python side

_array_priority_
 Array priority.

_array_struct_
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

```

nbytes
    length of item in bytes

ndim
    number of array dimensions

real
    real part of scalar

shape
    tuple of array dimensions

size
    number of elements in the gentype

strides
    tuple of bytes steps in each dimension

class broadcast(\_\_builtin\_\_.object)
    Produce an object that mimics broadcasting.

    Parameters
    -----
    in1, in2, ... : array_like
        Input parameters.

    Returns
    -----
    b : broadcast object
        Broadcast the input parameters against one another, and
        return an object that encapsulates the result.
        Amongst others, it has ``shape`` and ``nd`` properties, and
        may be used as an iterator.

    Examples
    -----
    Manually adding two vectors, using broadcasting:

    >>> x = np.array([[1], [2], [3]])
    >>> y = np.array([4, 5, 6])
    >>> b = np.broadcast(x, y)

    >>> out = np.empty(b.shape)
    >>> out.flat = [u+v for (u,v) in b]
    >>> out
    array([[ 5.,  6.,  7.],
           [ 6.,  7.,  8.],
           [ 7.,  8.,  9.]])]

    Compare against built-in broadcasting:

    >>> x + y
    array([[5, 6, 7],
           [6, 7, 8],
           [7, 8, 9]])]

    Methods defined here:

    \_\_iter\_\_\(...\)
        x.\_\_iter\_\_() <=> iter(x)

    next\(...\)
        x.next() -> the next value, or raise StopIteration

    reset\(...\)
        reset\(\)

        Reset the broadcasted result's iterator(s).

        Parameters
        -----
        None

        Returns
        -----
        None

        Examples
        -----
        >>> x = np.array([1, 2, 3])

```

```
>>> y = np.array([[4], [5], [6]]
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (2, 4), (3, 4))
>>> b.index
3
>>> b.reset()
>>> b.index
0
```

Data descriptors defined here:

index

current index in broadcasted result

Examples

```
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (1, 5), (1, 6))
>>> b.index
3
```

iters

tuple of iterators along ``self``'s "components."

Returns a tuple of `numpy.flatiter` objects, one for each "component" of ``self``.

See Also

`numpy.flatiter`

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> row, col = b.iters
>>> row.next(), col.next()
(1, 4)
```

nd

Number of dimensions of broadcasted result.

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.nd
2
```

numiter

Number of iterators possessed by the broadcasted result.

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.numiter
2
```

shape

Shape of broadcasted result.

Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.shape
(3, 3)
```

size

Total size of broadcasted result.

Examples

```
-----
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.size
9
```

Data and other attributes defined here:

__new__ = <built-in method __new__ of type object>
`T.__new__(S, ...) -> a new object with type S, a subtype of T`

class **busdaycalendar**(builtin object)

busdaycalendar(weekmask='1111100', holidays=None)

A business day calendar object that efficiently stores information defining valid days for the busday family of functions.

The default valid days are Monday through Friday ("business days"). A busdaycalendar object can be specified with any set of weekly valid days, plus an optional "holiday" dates that always will be invalid.

Once a busdaycalendar object is created, the weekmask and holidays cannot be modified.

.. versionadded:: 1.7.0

Parameters

```
-----
weekmask : str or array_like of bool, optional
    A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun
holidays : array_like of datetime64[D], optional
    An array of dates to consider as invalid dates, no matter which weekday they fall upon. Holiday dates may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.
```

Returns

```
-----
out : busdaycalendar
    A business day calendar object containing the specified weekmask and holidays values.
```

See Also

```
-----
is_busday : Returns a boolean array indicating valid days.
busday_offset : Applies an offset counted in valid days.
busday_count : Counts how many valid days are in a half-open date range.
```

Attributes

```
-----
Note: once a busdaycalendar object is created, you cannot modify the weekmask or holidays. The attributes return copies of internal data.
weekmask : (copy) seven-element array of bool
holidays : (copy) sorted array of datetime64[D]
```

Examples

```
-----
>>> # Some important days in July
... bdd = np.busdaycalendar(
...     holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
>>> # Default is Monday to Friday weekdays
... bdd.weekmask
array([ True,  True,  True,  True, False, False], dtype='bool')
>>> # Any holidays already on the weekend are removed
... bdd.holidays
array(['2011-07-01', '2011-07-04'], dtype='datetime64[D]')
```

Methods defined here:

__init__(...)
`x.__init__(...) initializes x; see help(type(x)) for signature`

Data descriptors defined here:

holidays

A copy of the `holiday` array indicating additional invalid days.

weekmask

A copy of the seven-element boolean mask indicating valid days.

Data and other attributes defined here:

`_new_` = <built-in method `_new_` of type object>
`T. new_(S, ...)` -> a new `object` with type `S`, a subtype of `T`

byte = class int8(signedinteger)

8-bit `integer`. Character code ``b''. C char compatible.

Method resolution order:

`int8`
`signedinteger`
`integer`
`number`
`generic`
`builtin object`

Methods defined here:

`_eq_(...)`
`x. eq_(y) <=> x==y`

`_ge_(...)`
`x. ge_(y) <=> x>=y`

`_gt_(...)`
`x. gt_(y) <=> x>y`

`_hash_(...)`
`x. hash_() <=> hash(x)`

`_index_(...)`
`x[y:z] <=> x[y. index_():z. index_()]`

`_le_(...)`
`x. le_(y) <=> x<=y`

`_lt_(...)`
`x. lt_(y) <=> x<y`

`_ne_(...)`
`x. ne_(y) <=> x!=y`

Data and other attributes defined here:

`_new_` = <built-in method `_new_` of type object>
`T. new_(S, ...)` -> a new `object` with type `S`, a subtype of `T`

Data descriptors inherited from `integer`:

denominator

denominator of value (1)

numerator

numerator of value (the value itself)

Methods inherited from `generic`:

`_abs_(...)`
`x. abs_() <=> abs(x)`

`_add_(...)`
`x. add_(y) <=> x+y`

```
__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
```

```
x.rand(y) <==> y&x
rdiv_(...)
x.rdiv(y) <==> y/x
rdivmod_(...)
x.rdivmod(y) <==> divmod(y, x)
reduce_(...)
repr_(...)
x.repr() <==> repr(x)
rfloordiv_(...)
x.rfloordiv(y) <==> y//x
rlshift_(...)
x.rlshift(y) <==> y<<x
rmod_(...)
x.rmod(y) <==> y%x
rmul_(...)
x.rmul(y) <==> y*x
ror_(...)
x.ror(y) <==> y|x
rpow_(...)
y.rpow(x[, z]) <==> pow(x, y[, z])
rrshift_(...)
x.rrshift(y) <==> y>>x
rshift_(...)
x.rshift(y) <==> x>>y
rsub_(...)
x.rsub(y) <==> y-x
rtruediv_(...)
x.rtruediv(y) <==> y/x
rxor_(...)
x.rxor(y) <==> y^x
setstate_(...)
sizeof_(...)
str_(...)
x.str() <==> str(x)
sub_(...)
x.sub(y) <==> x-y
truediv_(...)
x.truediv(y) <==> x/y
xor_(...)
x.xor(y) <==> x^y
all(...)
Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)

[newbyteorder](#)(new_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

- * 'S' - swap [dtype](#) from current to opposite endian
- * {'<', 'L'} - little endian
- * {'>', 'B'} - big endian
- * {'=', 'N'} - native order
- * {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

`new_order : str`, optional

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

nonzero(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.
```

Data descriptors inherited from [generic](#):

T
 transpose

_array_interface_
 Array protocol: Python side

_array_priority_
 Array priority.

_array_struct_
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
get array data-descriptor

flags
integer value of flags

flat
a 1-d view of scalar

imag
imaginary part of scalar

itemsize
length of one element in bytes

nbytes
length of item in bytes

ndim
number of array dimensions

real
real part of scalar

shape
tuple of array dimensions

size
number of elements in the gentype

strides
tuple of bytes steps in each dimension

bytes_ = class string_([__builtin__.str](#), [character](#))

Method resolution order:

[string](#)
[__builtin__.str](#)
[__builtin__.basestring](#)
[character](#)
[flexible](#)
[generic](#)
[__builtin__.object](#)

Methods defined here:

[__eq__\(...\)](#)
x.[__eq__](#)(y) <==> x==y

[__ge__\(...\)](#)
x.[__ge__](#)(y) <==> x>=y

[__gt__\(...\)](#)
x.[__gt__](#)(y) <==> x>y

[__hash__\(...\)](#)
x.[__hash__](#)() <==> hash(x)

[__le__\(...\)](#)
x.[__le__](#)(y) <==> x<=y

[__lt__\(...\)](#)
x.[__lt__](#)(y) <==> x<y

[__ne__\(...\)](#)
x.[__ne__](#)(y) <==> x!=y

[__repr__\(...\)](#)
x.[__repr__](#)() <==> repr(x)

[__str__\(...\)](#)

x.str() <==> str(x)

Data and other attributes defined here:

new = <built-in method new of type object>
T.new(S, ...) -> a new object with type S, a subtype of T

Methods inherited from builtin.str:

add(...)
x.add(y) <==> x+y

contains(...)
x.contains(y) <==> y in x

format(...)
S.format(format_spec) -> string
Return a formatted version of S as described by format_spec.

getattribute(...)
x.getattribute('name') <==> x.name

getitem(...)
x.getitem(y) <==> x[y]

getnewargs(...)

getslice(...)
x.getslice(i, j) <==> x[i:j]
Use of negative indices is not supported.

len(...)
x.len() <==> len(x)

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(n) <==> x*n

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(n) <==> n*x

sizeof(...)
S.sizeof() -> size of S in memory, in bytes

capitalize(...)
S.capitalize() -> string
Return a copy of the string S with only its first character capitalized.

center(...)
S.center(width[, fillchar]) -> string
Return S centered in a string of length width. Padding is done using the specified fill character (default is a space)

count(...)
S.count(sub[, start[, end]]) -> int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode(...)
S.decode([encoding[,errors]]) -> object
Decodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise

a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that is able to handle UnicodeDecodeErrors.

encode(...)

S.encode([encoding[,errors]]) -> object

Encodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that is able to handle UnicodeEncodeErrors.

endswith(...)

S.endswith(suffix[, start[, end]]) -> bool

Return True if S ends with the specified suffix, False otherwise.
With optional start, test S beginning at that position.
With optional end, stop comparing S at that position.
suffix can also be a tuple of strings to try.

expandtabs(...)

S.expandtabs([tabsize]) -> string

Return a copy of S where all tab characters are expanded using spaces.
If tabsize is not given, a tab size of 8 characters is assumed.

find(...)

S.find(sub [,start [,end]]) -> int

Return the lowest index in S where substring sub is found,
such that sub is contained within S[start:end]. Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(...)

S.format(*args, **kwargs) -> string

Return a formatted version of S, using substitutions from args and kwargs.
The substitutions are identified by braces ('{' and '}').

index(...)

S.index(sub [,start [,end]]) -> int

Like S.find() but raise ValueError when the substring is not found.

isalnum(...)

S.isalnum() -> bool

Return True if all characters in S are alphanumeric
and there is at least one character in S, False otherwise.

isalpha(...)

S.isalpha() -> bool

Return True if all characters in S are alphabetic
and there is at least one character in S, False otherwise.

isdigit(...)

S.isdigit() -> bool

Return True if all characters in S are digits
and there is at least one character in S, False otherwise.

islower(...)

S.islower() -> bool

Return True if all cased characters in S are lowercase and there is
at least one cased character in S, False otherwise.

isspace(...)

S.isspace() -> bool

Return True if all characters in S are whitespace
and there is at least one character in S, False otherwise.

istitle(...)

S.istitle() -> bool

Return True if S is a titlecased string and there is at least one

`character` in S, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper(...)
`S.isupper() -> bool`

Return True if all cased characters in S are uppercase and there is at least one cased `character` in S, False otherwise.

join(...)
`S.join(iterable) -> string`

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust(...)
`S.ljust(width[, fillchar]) -> string`

Return S left-justified in a string of length width. Padding is done using the specified fill `character` (default is a space).

lower(...)
`S.lower() -> string`

Return a copy of the string S converted to lowercase.

lstrip(...)
`S.lstrip([chars]) -> string or unicode`

Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is `unicode`, S will be converted to `unicode` before stripping

partition(...)
`S.partition(sep) -> (head, sep, tail)`

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace(...)
`S.replace(old, new[, count]) -> string`

Return a copy of string S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind(...)
`S.rfind(sub [,start [,end]]) -> int`

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(...)
`S.rindex(sub [,start [,end]]) -> int`

Like S.`rfind()` but raise ValueError when the substring is not found.

rjust(...)
`S.rjust(width[, fillchar]) -> string`

Return S right-justified in a string of length width. Padding is done using the specified fill `character` (default is a space)

rpartition(...)
`S.rpartition(sep) -> (head, sep, tail)`

Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit(...)
`S.rsplit([sep [,maxsplit]]) -> list of strings`

Return a list of the words in the string S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string

is a separator.

rstrip(...)
S.rstrip([chars]) -> string or unicode
Return a copy of the string S with trailing whitespace removed.
If chars is given and not None, remove characters in chars instead.
If chars is unicode, S will be converted to unicode before stripping

split(...)
S.split([sep [,maxsplit]]) -> list of strings
Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines(...)
S.splitlines(keepends=False) -> list of strings
Return a list of the lines in S, breaking at line boundaries.
Line breaks are not included in the resulting list unless keepends is given and true.

startswith(...)
S.startswith(prefix[, start[, end]]) -> bool
Return True if S starts with the specified prefix, False otherwise.
With optional start, test S beginning at that position.
With optional end, stop comparing S at that position.
prefix can also be a tuple of strings to try.

strip(...)
S.strip([chars]) -> string or unicode
Return a copy of the string S with leading and trailing whitespace removed.
If chars is given and not None, remove characters in chars instead.
If chars is unicode, S will be converted to unicode before stripping

swapcase(...)
S.swapcase() -> string
Return a copy of the string S with uppercase characters converted to lowercase and vice versa.

title(...)
S.title() -> string
Return a titlecased version of S, i.e. words start with uppercase characters, all remaining cased characters have lowercase.

translate(...)
S.translate(table [,deletechars]) -> string
Return a copy of the string S, where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256 or None. If the table argument is None, no translation is applied and the operation simply removes the characters in deletechars.

upper(...)
S.upper() -> string
Return a copy of the string S converted to uppercase.

zfill(...)
S.zfill(width) -> string
Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

Methods inherited from [generic](#):

__abs__(...)
x.abs() <==> abs(x)

__and__(...)

```
x. and_(y) <==> x&y  
array_(...)  
    sc. array_(|type) return 0-dim array  
array_wrap_(...)  
    sc. array_wrap_(obj) return scalar from array  
copy_(...)  
deepcopy_(...)  
div_(...)  
    x. div_(y) <==> x/y  
divmod_(...)  
    x. divmod_(y) <==> divmod(x, y)  
float_(...)  
    x. float_( ) <==> float(x)  
floordiv_(...)  
    x. floordiv_(y) <==> x//y  
hex_(...)  
    x. hex_( ) <==> hex(x)  
int_(...)  
    x. int_( ) <==> int(x)  
invert_(...)  
    x. invert_( ) <==> ~x  
long_(...)  
    x. long_( ) <==> long(x)  
lshift_(...)  
    x. lshift_(y) <==> x<<y  
neg_(...)  
    x. neg_( ) <==> -x  
nonzero_(...)  
    x. nonzero_( ) <==> x != 0  
oct_(...)  
    x. oct_( ) <==> oct(x)  
or_(...)  
    x. or_(y) <==> x|y  
pos_(...)  
    x. pos_( ) <==> +x  
pow_(...)  
    x. pow_(y[, z]) <==> pow(x, y[, z])  
radd_(...)  
    x. radd_(y) <==> y+x  
rand_(...)  
    x. rand_(y) <==> y&x  
rdiv_(...)  
    x. rdiv_(y) <==> y/x  
rdivmod_(...)  
    x. rdivmod_(y) <==> divmod(y, x)  
reduce_(...)  
rfloordiv_(...)  
    x. rfloordiv_(y) <==> y//x
```

```
__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

    Class generic exists solely to derive numpy scalars from, and possesses,
    albeit unimplemented, all the attributes of the ndarray class
    so as to provide a uniform API.

    See Also
    -----
    The corresponding attribute of the derived class of interest.
```

argsort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)

[newbyteorder](#)(new_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap [dtype](#) from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first

letter of `new_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

`new_dtype : dtype`
 New `'dtype'` object with the given change to the `byte` order.

nonzero(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```
-----
The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

toString(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

Data descriptors inherited from [generic](#):

T
transpose

__array_interface__
Array protocol: Python side

__array_priority__
Array priority.

__array_struct__
Array protocol: struct

base
base object

data
pointer to start of data

dtype
get array data-descriptor

flags
integer value of flags

flat
a 1-d view of scalar

imag
imaginary part of scalar

itemsize
length of one element in bytes

nbytes
length of item in bytes

ndim
number of array dimensions

real
real part of scalar

shape
tuple of array dimensions

size

number of elements in the gentype

strides

tuple of bytes steps in each dimension

cdouble = class complex128([complexfloating](#), [builtin .complex](#))

Composed of two 64 bit floats

Method resolution order:

[complex128](#)
[complexfloating](#)
[inexact](#)
[number](#)
[generic](#)
[builtin .complex](#)
[builtin .object](#)

Methods defined here:

[__eq__\(...\)](#)
x. [__eq__\(y\)](#) <==> x==y

[__ge__\(...\)](#)
x. [__ge__\(y\)](#) <==> x>=y

[__gt__\(...\)](#)
x. [__gt__\(y\)](#) <==> x>y

[__le__\(...\)](#)
x. [__le__\(y\)](#) <==> x<=y

[__lt__\(...\)](#)
x. [__lt__\(y\)](#) <==> x<y

[__ne__\(...\)](#)
x. [__ne__\(y\)](#) <==> x!=y

[__repr__\(...\)](#)
x. [__repr__\(\)](#) <==> repr(x)

[__str__\(...\)](#)
x. [__str__\(\)](#) <==> [str\(x\)](#)

Data and other attributes defined here:

[__new__](#) = <built-in method [__new__](#) of type object>
T. [__new__\(S, ...\)](#) -> a new [object](#) with type S, a subtype of T

Methods inherited from [generic](#):

[__abs__\(...\)](#)
x. [__abs__\(\)](#) <==> abs(x)

[__add__\(...\)](#)
x. [__add__\(y\)](#) <==> x+y

[__and__\(...\)](#)
x. [__and__\(y\)](#) <==> x&y

[__array__\(...\)](#)
sc. [__array__\(|type\)](#) return 0-dim array

[__array_wrap__\(...\)](#)
sc. [__array_wrap__\(obj\)](#) return scalar from array

[__copy__\(...\)](#)

[__deepcopy__\(...\)](#)

```
_div_(...)
x._div_(y) <==> x/y

_divmod_(...)
x._divmod_(y) <==> divmod(x, y)

_float_(...)
x._float_() <==> float(x)

_floordiv_(...)
x._floordiv_(y) <==> x//y

_format_(...)
NumPy array scalar formatter

_getitem_(...)
x._getitem_(y) <==> x[y]

_hex_(...)
x._hex_() <==> hex(x)

_int_(...)
x._int_() <==> int(x)

_invert_(...)
x._invert_() <==> ~x

_long_(...)
x._long_() <==> long(x)

_lshift_(...)
x._lshift_(y) <==> x<<y

_mod_(...)
x._mod_(y) <==> x%y

_mul_(...)
x._mul_(y) <==> x*y

_neg_(...)
x._neg_() <==> -x

_nonzero_(...)
x._nonzero_() <==> x != 0

_oct_(...)
x._oct_() <==> oct(x)

_or_(...)
x._or_(y) <==> x|y

_pos_(...)
x._pos_() <==> +x

_pow_(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

_radd_(...)
x._radd_(y) <==> y+x

_rand_(...)
x._rand_(y) <==> y&x

_rdiv_(...)
x._rdiv_(y) <==> y/x

_rdivmod_(...)
x._rdivmod_(y) <==> divmod(y, x)

_reduce_(...)

_rfloordiv_(...)
x._rfloordiv_(y) <==> y//x
```

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
```

```
* {'=': 'N'} - native order
* {'|': 'I'} - ignore (no change to byte order)

Parameters
-----
new_order : str, optional
    Byte order to force; a value from the byte order specifications
    above. The default value ('S') results in swapping the current
    byte order. The code does a case-insensitive check on the first
    letter of `new_order` for the alternatives above. For example,
    any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns
-----
new_dtype : dtype
    New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

reshape(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

Data descriptors inherited from [generic](#):

T

transpose

__array_interface__

Array protocol: Python side

__array_priority__

Array priority.

__array_struct__

Array protocol: struct

base

base object

data

pointer to start of data

dtype

get array data-descriptor

flags

integer value of flags

flat

a 1-d view of scalar

imag

imaginary part of scalar

itemsize

length of one element in bytes

nbytes

length of item in bytes

ndim
number of array dimensions

real
real part of scalar

shape
tuple of array dimensions

size
number of elements in the gentype

strides
tuple of bytes steps in each dimension

Methods inherited from [builtin_complex](#):

__coerce__(...)
x. [coerce](#)(y) <==> coerce(x, y)

__getattribute__(...)
x. [getattribute](#)('name') <==> x.name

__getnewargs__(...)

__hash__(...)
x. [hash](#)() <==> hash(x)

cfloat = class complex128([complexfloating](#), [builtin_complex](#))
Composed of two 64 bit floats

Method resolution order:

[complex128](#)
[complexfloating](#)
[inexact](#)
[number](#)
[generic](#)
[builtin_complex](#)
[builtin_object](#)

Methods defined here:

__eq__(...)
x. [eq](#)(y) <==> x==y

__ge__(...)
x. [ge](#)(y) <==> x>=y

__gt__(...)
x. [gt](#)(y) <==> x>y

__le__(...)
x. [le](#)(y) <==> x<=y

__lt__(...)
x. [lt](#)(y) <==> x<y

__ne__(...)
x. [ne](#)(y) <==> x!=y

__repr__(...)
x. [repr](#)() <==> repr(x)

__str__(...)
x. [str](#)() <==> [str](#)(x)

Data and other attributes defined here:

__new__ = <built-in method [new](#) of type object>

T.[new](#)(S, ...) -> a new [object](#) with type S, a subtype of T

Methods inherited from [generic](#):

[abs](#)(...)
x.[abs](#)() <==> abs(x)

[add](#)(...)
x.[add](#)(y) <==> x+y

[and](#)(...)
x.[and](#)(y) <==> x&y

[array](#)(...)
sc.[array](#)(|type) return 0-dim array

[array_wrap](#)(...)
sc.[array_wrap](#)(obj) return scalar from array

[copy](#)(...)

[deepcopy](#)(...)

[div](#)(...)
x.[div](#)(y) <==> x/y

[divmod](#)(...)
x.[divmod](#)(y) <==> divmod(x, y)

[float](#)(...)
x.[float](#)() <==> float(x)

[floordiv](#)(...)
x.[floordiv](#)(y) <==> x//y

[format](#)(...)
NumPy array scalar formatter

[getitem](#)(...)
x.[getitem](#)(y) <==> x[y]

[hex](#)(...)
x.[hex](#)() <==> hex(x)

[int](#)(...)
x.[int](#)() <==> int(x)

[invert](#)(...)
x.[invert](#)() <==> ~x

[long](#)(...)
x.[long](#)() <==> long(x)

[lshift](#)(...)
x.[lshift](#)(y) <==> x<<y

[mod](#)(...)
x.[mod](#)(y) <==> x%y

[mul](#)(...)
x.[mul](#)(y) <==> x*y

[neg](#)(...)
x.[neg](#)() <==> -x

[nonzero](#)(...)
x.[nonzero](#)() <==> x != 0

[oct](#)(...)
x.[oct](#)() <==> oct(x)

[or](#)(...)
x.[or](#)(y) <==> x|y

```
__pos__(...)  
x.__pos__() <==> +x  
  
__pow__(...)  
x.__pow__(y[, z]) <==> pow(x, y[, z])  
  
__radd__(...)  
x.__radd__(y) <==> y+x  
  
__rand__(...)  
x.__rand__(y) <==> y&x  
  
__rdiv__(...)  
x.__rdiv__(y) <==> y/x  
  
__rdivmod__(...)  
x.__rdivmod__(y) <==> divmod(y, x)  
  
__reduce__(...)  
  
__rfloordiv__(...)  
x.__rfloordiv__(y) <==> y//x  
  
__rlshift__(...)  
x.__rlshift__(y) <==> y<<x  
  
__rmod__(...)  
x.__rmod__(y) <==> y%x  
  
__rmul__(...)  
x.__rmul__(y) <==> y*x  
  
__ror__(...)  
x.__ror__(y) <==> y|x  
  
__rpow__(...)  
y.__rpow__(x[, z]) <==> pow(x, y[, z])  
  
__rrshift__(...)  
x.__rrshift__(y) <==> y>>x  
  
__rshift__(...)  
x.__rshift__(y) <==> x>>y  
  
__rsub__(...)  
x.__rsub__(y) <==> y-x  
  
__rtruediv__(...)  
x.__rtruediv__(y) <==> y/x  
  
__rxor__(...)  
x.__rxor__(y) <==> y^x  
  
__setstate__(...)  
  
__sizeof__(...)  
  
__sub__(...)  
x.__sub__(y) <==> x-y  
  
__truediv__(...)  
x.__truediv__(y) <==> x/y  
  
__xor__(...)  
x.__xor__(y) <==> x^y  
  
all(...)  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses,  
albeit unimplemented, all the attributes of the ndarray class  
so as to provide a uniform API.  
  
See Also  
-----
```

The corresponding attribute of the derived class of interest.

any(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```
-----  
The corresponding attribute of the derived class of interest.  
min(...)  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses,  
albeit unimplemented, all the attributes of the ndarray class  
so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.  
  
newbyteorder(...)  
newbyteorder(new_order='S')  
  
Return a new `dtype` with a different byte order.  
  
Changes are also made in all fields and sub-arrays of the data type.  
  
The `new_order` code can be any from the following:  
  
* 'S' - swap dtype from current to opposite endian  
* {'<', 'L'} - little endian  
* {'>', 'B'} - big endian  
* {'=', 'N'} - native order  
* {'|', 'I'} - ignore (no change to byte order)  
  
Parameters  
-----  
new_order : str, optional  
    Byte order to force; a value from the byte order specifications  
    above. The default value ('S') results in swapping the current  
    byte order. The code does a case-insensitive check on the first  
    letter of `new_order` for the alternatives above. For example,  
    any of 'B' or 'b' or 'biggish' are valid to specify big-endian.  
  
Returns  
-----  
new_dtype : dtype  
    New `dtype` object with the given change to the byte order.  
  
nonzero(...)  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses,  
albeit unimplemented, all the attributes of the ndarray class  
so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.  
  
prod(...)  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses,  
albeit unimplemented, all the attributes of the ndarray class  
so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.  
  
ptp(...)  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses,  
albeit unimplemented, all the attributes of the ndarray class  
so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.  
  
put(...)  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses,  
albeit unimplemented, all the attributes of the ndarray class  
so as to provide a uniform API.  
  
See Also  
-----
```

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to

provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

Data descriptors inherited from [generic](#):

T
 transpose

__array_interface__
 Array protocol: Python side

__array_priority__
 Array priority.

__array_struct__
 Array protocol: struct

base

```
base object

data
    pointer to start of data

dtype
    get array data-descriptor

flags
    integer value of flags

flat
    a 1-d view of scalar

imag
    imaginary part of scalar

itemsize
    length of one element in bytes

nbytes
    length of item in bytes

ndim
    number of array dimensions

real
    real part of scalar

shape
    tuple of array dimensions

size
    number of elements in the gentype

strides
    tuple of bytes steps in each dimension
```

Methods inherited from [__builtin__.complex](#):

```
__coerce__(...)
    x. \_\_coerce\_\_ (y) <==> coerce(x, y)

__getattribute__(...)
    x. \_\_getattribute\_\_ ('name') <==> x.name

__getnewargs__(...)

__hash__(...)
    x. \_\_hash\_\_ () <==> hash(x)
```

class **character**([flexible](#))

Method resolution order:

```
character
flexible
generic
\_\_builtin\_\_.object
```

Methods inherited from [generic](#):

```
__abs__(...)
    x. \_\_abs\_\_ () <==> abs(x)

__add__(...)
    x. \_\_add\_\_ (y) <==> x+y

__and__(...)
    x. \_\_and\_\_ (y) <==> x&y

__array__(...)
```

```
sc.array(|type) return 0-dim array
array_wrap(...)
sc.array_wrap(obj) return scalar from array
copy(...)
deepcopy(...)
div(...)
x.div(y) <==> x/y
divmod(...)
x.divmod(y) <==> divmod(x, y)
eq(...)
x.eq(y) <==> x==y
float(...)
x.float() <==> float(x)
floordiv(...)
x.floordiv(y) <==> x//y
format(...)
NumPy array scalar formatter
ge(...)
x.ge(y) <==> x>=y
getitem(...)
x.getitem(y) <==> x[y]
gt(...)
x.gt(y) <==> x>y
hex(...)
x.hex() <==> hex(x)
int(...)
x.int() <==> int(x)
invert(...)
x.invert() <==> ~x
le(...)
x.le(y) <==> x<=y
long(...)
x.long() <==> long(x)
lshift(...)
x.lshift(y) <==> x<<y
lt(...)
x.lt(y) <==> x<y
mod(...)
x.mod(y) <==> x%y
mul(...)
x.mul(y) <==> x*y
ne(...)
x.ne(y) <==> x!=y
neg(...)
x.neg() <==> -x
nonzero(...)
x.nonzero() <==> x != 0
oct(...)
x.oct() <==> oct(x)
```

```
__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y
```

all(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```
-----
The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also
-----
The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

fill(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

flatten(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
[newbyteorder](#)(new_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

- * 'S' - swap [dtype](#) from current to opposite endian
- * {'<', 'L'} - little endian
- * {'>', 'B'} - big endian
- * {'=', 'N'} - native order
- * {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

`new_order : str, optional`
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

`new_dtype : dtype`
New `dtype` [object](#) with the given change to the [byte](#) order.

nonzero(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setflags(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sort(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

squeeze(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

Data descriptors inherited from [generic](#):

T
 transpose

_array_interface_
 Array protocol: Python side

```

__array_priority__
    Array priority.

__array_struct__
    Array protocol: struct

base
    base object

data
    pointer to start of data

dtype
    get array data-descriptor

flags
    integer value of flags

flat
    a 1-d view of scalar

imag
    imaginary part of scalar

itemsize
    length of one element in bytes

nbytes
    length of item in bytes

ndim
    number of array dimensions

real
    real part of scalar

shape
    tuple of array dimensions

size
    number of elements in the gentype

strides
    tuple of bytes steps in each dimension

class chararray(numpy.ndarray)
    chararray(shape, itemsize=1, unicode=False, buffer=None, offset=0,
               strides=None, order=None)

    Provides a convenient view on arrays of string and unicode values.

    .. note::
        The 'chararray' class exists for backwards compatibility with
        Numarray, it is not recommended for new development. Starting from numpy
        1.4, if one needs arrays of strings, it is recommended to use arrays of
        'dtype', 'object', 'string' or 'unicode', and use the free functions
        in the 'numpy.char' module for fast vectorized string operations.

    Versus a regular Numpy array of type 'str' or 'unicode', this
    class adds the following functionality:

    1) values automatically have whitespace removed from the end
       when indexed

    2) comparison operators automatically remove whitespace from the
       end when comparing values

    3) vectorized string operations are provided as methods
       (e.g. '.endswith') and infix operators (e.g. ``+``, ``*``, ``%``)

    chararrays should be created using 'numpy.char.array' or
    'numpy.char.asarray', rather than this constructor directly.

    This constructor creates the array, using 'buffer' (with 'offset'
    and 'strides') if it is not 'None'. If 'buffer' is 'None', then
    constructs a new array with 'strides' in "C order", unless both
    'len(shape) >= 2' and 'order='Fortran'', in which case 'strides'

```

```
is in "Fortran order".  
Methods  
-----  
astype  
argsort  
copy  
count  
decode  
dump  
dumps  
encode  
endswith  
expandtabs  
fill  
find  
flatten  
index  
isalnum  
isalpha  
isdecimal  
isdigit  
islower  
isnumeric  
isspace  
istitle  
isupper  
item  
join  
ljust  
lower  
lstrip  
nonzero  
put  
ravel  
repeat  
replace  
reshape  
resize  
rfind  
rindex  
rjust  
rsplit  
rstrip  
searchsorted  
setfield  
setflags  
sort  
split  
splitlines  
squeeze  
startswith  
strip  
swapaxes  
swapcase  
take  
title  
tofile  
tolist  
tostring  
translate  
transpose  
upper  
view  
zfill  
  
Parameters  
-----  
shape : tuple  
    Shape of the array.  
itemsize : int, optional  
    Length of each array element, in number of characters. Default is 1.  
unicode : bool, optional  
    Are the array elements of type unicode (True) or string (False).  
    Default is False.  
buffer : int, optional  
    Memory address of the start of the array data. Default is None,  
    in which case a new array is created.  
offset : int, optional  
    Fixed stride displacement from the beginning of an axis?  
    Default is 0. Needs to be >=0.  
strides : array_like of ints, optional  
    Strides for the array (see `ndarray.strides` for full description).  
    Default is None.  
order : {'C', 'F'}, optional  
    The order in which the array data is stored in memory: 'C' ->  
    "row major" order (the default), 'F' -> "column major"  
    (Fortran) order.
```

Examples

```
-----
>>> charar = np.chararray((3, 3))
>>> charar[:] = 'a'
>>> charar
chararray([['a', 'a', 'a'],
           ['a', 'a', 'a'],
           ['a', 'a', 'a']],
          dtype='|S1')

>>> charar = np.chararray(charar.shape, itemsize=5)
>>> charar[:] = 'abc'
>>> charar
chararray([['abc', 'abc', 'abc'],
           ['abc', 'abc', 'abc'],
           ['abc', 'abc', 'abc']],
          dtype='|S5')
```

Method resolution order:

```
chararray
numpy.ndarray
builtin_object
```

Methods defined here:**__add__(self, other)**

Return (self + other), that is string concatenation,
element-wise for a pair of array_likes of [str](#) or [unicode](#).

See also

add

__array_finalize__(self, obj)**__eq__(self, other)**

Return (self == other) element-wise.

See also

equal

__ge__(self, other)

Return (self >= other) element-wise.

See also

greater_equal

__getitem__(self, obj)**__gt__(self, other)**

Return (self > other) element-wise.

See also

greater

__le__(self, other)

Return (self <= other) element-wise.

See also

less_equal

__lt__(self, other)

Return (self < other) element-wise.

See also

less

__mod__(self, i)

Return (self % i), that is pre-Python 2.6 string formatting
(interpolation), element-wise for a pair of array_likes of [string](#) `or [unicode](#) `.

See also

mod

```
__mul__(self, i)
    Return (self * i), that is string multiple concatenation,
    element-wise.

    See also
    -----
    multiply

__ne__(self, other)
    Return (self != other) element-wise.

    See also
    -----
    not_equal

__radd__(self, other)
    Return (other + self), that is string concatenation,
    element-wise for a pair of array_likes of `string` or `unicode`.

    See also
    -----
    add

__rmod__(self, other)

__rmul__(self, i)
    Return (self * i), that is string multiple concatenation,
    element-wise.

    See also
    -----
    multiply

argsort(self, axis=-1, kind='quicksort', order=None)

capitalize(self)
    Return a copy of `self` with only the first character of each element
    capitalized.

    See also
    -----
    char.capitalize

center(self, width, fillchar=' ')
    Return a copy of `self` with its elements centered in a
    string of length `width`.

    See also
    -----
    center

count(self, sub, start=0, end=None)
    Returns an array with the number of non-overlapping occurrences of
    substring `sub` in the range [`start`, `end`].

    See also
    -----
    char.count

decode(self, encoding=None, errors=None)
    Calls 'str'.decode' element-wise.

    See also
    -----
    char.decode

encode(self, encoding=None, errors=None)
    Calls 'str.encode' element-wise.

    See also
    -----
    char.encode

endswith(self, suffix, start=0, end=None)
    Returns a boolean array which is `True` where the string element
    in `self` ends with `suffix`, otherwise `False`.

    See also
    -----
    char.endswith

expandtabs(self, tabsize=8)
```

Return a copy of each string element where all tab characters are replaced by one or more spaces.

See also

`char.expandtabs`

find(self, sub, start=0, end=None)
For each element, return the lowest index in the string where substring `sub` is found.

See also

`char.find`

index(self, sub, start=0, end=None)
Like `find`, but raises `ValueError` when the substring is not found.

See also

`char.index`

isalnum(self)
Returns true for each element if all characters in the string are alphanumeric and there is at least one [character](#), false otherwise.

See also

`char.isalnum`

isalpha(self)
Returns true for each element if all characters in the string are alphabetic and there is at least one [character](#), false otherwise.

See also

`char.isalpha`

isdecimal(self)
For each element in `self`, return True if there are only decimal characters in the element.

See also

`char.isdecimal`

isdigit(self)
Returns true for each element if all characters in the string are digits and there is at least one [character](#), false otherwise.

See also

`char.isdigit`

islower(self)
Returns true for each element if all cased characters in the string are lowercase and there is at least one cased [character](#), false otherwise.

See also

`char.islower`

isnumeric(self)
For each element in `self`, return True if there are only numeric characters in the element.

See also

`char.isnumeric`

isspace(self)
Returns true for each element if there are only whitespace characters in the string and there is at least one [character](#), false otherwise.

See also

`char.isspace`

istitle(self)

Returns true for each element if the element is a titlecased string and there is at least one [character](#), false otherwise.

See also

`char.istitle`

isupper(self)
Returns true for each element if all cased characters in the string are uppercase and there is at least one [character](#), false otherwise.

See also

`char.isupper`

join(self, seq)
Return a string which is the concatenation of the strings in the sequence `seq`.

See also

`char.join`

ljust(self, width, fillchar=' ')
Return an array with the elements of `self` left-justified in a string of length `width`.

See also

`char.ljust`

lower(self)
Return an array with the elements of `self` converted to lowercase.

See also

`char.lower`

lstrip(self, chars=None)
For each element in `self`, return a copy with the leading characters removed.

See also

`char.lstrip`

partition(self, sep)
Partition each element in `self` around `sep`.

See also

`partition`

replace(self, old, new, count=None)
For each element in `self`, return a copy of the string with all occurrences of substring `old` replaced by `new`.

See also

`char.replace`

rfind(self, sub, start=0, end=None)
For each element in `self`, return the highest index in the string where substring `sub` is found, such that `sub` is contained within [`start`, `end`].

See also

`char.rfind`

rindex(self, sub, start=0, end=None)
Like `rfind`, but raises `ValueError` when the substring `sub` is not found.

See also

`char.rindex`

rjust(self, width, fillchar=' ')
Return an array with the elements of `self` right-justified in a string of length `width`.

```
See also
-----
char.rjust

rpartition(self, sep)
    Partition each element in `self` around `sep`.

    See also
    -----
    rpartition

rsplit(self, sep=None, maxsplit=None)
    For each element in `self`, return a list of the words in
    the string, using `sep` as the delimiter string.

    See also
    -----
    char.rsplit

rstrip(self, chars=None)
    For each element in `self`, return a copy with the trailing
    characters removed.

    See also
    -----
    char.rstrip

split(self, sep=None, maxsplit=None)
    For each element in `self`, return a list of the words in the
    string, using `sep` as the delimiter string.

    See also
    -----
    char.split

splitlines(self, keepends=None)
    For each element in `self`, return a list of the lines in the
    element, breaking at line boundaries.

    See also
    -----
    char.splitlines

startswith(self, prefix, start=0, end=None)
    Returns a boolean array which is `True` where the string element
    in `self` starts with `prefix`, otherwise `False`.

    See also
    -----
    char.startswith

strip(self, chars=None)
    For each element in `self`, return a copy with the leading and
    trailing characters removed.

    See also
    -----
    char.strip

swapcase(self)
    For each element in `self`, return a copy of the string with
    uppercase characters converted to lowercase and vice versa.

    See also
    -----
    char.swapcase

title(self)
    For each element in `self`, return a titlecased version of the
    string: words start with uppercase characters, all remaining cased
    characters are lowercase.

    See also
    -----
    char.title

translate(self, table, deletechars=None)
    For each element in `self`, return a copy of the string where
    all characters occurring in the optional argument
    `deletechars` are removed, and the remaining characters have
    been mapped through the given translation table.
```

See also

`char.translate`

upper(self)
Return an array with the elements of `self` converted to uppercase.

See also

`char.upper`

zfill(self, width)
Return the numeric string left-filled with zeros in a string of length `width`.

See also

`char.zfill`

Static methods defined here:

__new__(subtype, shape, itemsize=1, unicode=False, buffer=None, offset=0, strides=None, order='C')

Data descriptors defined here:

__dict__
dictionary for instance variables (if defined)

Methods inherited from [numpy.ndarray](#):

__abs__(...)
`x. __abs__()` <==> `abs(x)`

__and__(...)
`x. __and__(y)` <==> `x&y`

__array__(...)
`a. __array__(|dtype)` -> reference if type unchanged, copy otherwise.
Returns either a new reference to self if `dtype` is not given or a new array of provided data type if `dtype` is different from the current `dtype` of the array.

__array_prepare__(...)
`a. __array_prepare__(obj)` -> Object of same type as [ndarray object](#) obj.

__array_wrap__(...)
`a. __array_wrap__(obj)` -> Object of same type as [ndarray object](#) a.

__contains__(...)
`x. __contains__(y)` <==> `y in x`

__copy__(...)
`a. __copy__([order])`
Return a copy of the array.

Parameters

`order : {'C', 'F', 'A'}, optional`
If order is 'C' (False) then the result is contiguous (default).
If order is 'Fortran' (True) then the result has fortran order.
If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

__deepcopy__(...)
`a. __deepcopy__()` -> Deep copy of array.

Used if `copy.deepcopy` is called on an array.

__delitem__(...)
`x. __delitem__(y)` <==> `del x[y]`

__delslice__(...)
`x. __delslice__(i, j)` <==> `del x[i:j]`

```
Use of negative indices is not supported.

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__getslice__(...)
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

__hex__(...)
x.__hex__() <==> hex(x)

__iadd__(...)
x.__iadd__(y) <==> x+=y

__iand__(...)
x.__iand__(y) <==> x&=y

__idiv__(...)
x.__idiv__(y) <==> x/=y

__ifloordiv__(...)
x.__ifloordiv__(y) <==> x//=y

__ilshift__(...)
x.__ilshift__(y) <==> x<<=y

__imod__(...)
x.__imod__(y) <==> x%>=y

__imul__(...)
x.__imul__(y) <==> x*>=y

__index__(...)
x[y:z] <==> x[y.__index__():z.__index__()]

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__ior__(...)
x.__ior__(y) <==> x|=y

__ipow__(...)
x.__ipow__(y) <==> x**=y

__irshift__(...)
x.__irshift__(y) <==> x>>=y

__isub__(...)
x.__isub__(y) <==> x-=y

__iter__(...)
x.__iter__() <==> iter(x)

__itruediv__(...)
x.__itruediv__(y) <==> x/=y

__ixor__(...)
x.__ixor__(y) <==> x^=y

__len__(...)
```

```
x. len () <==> len(x)

long (...)  
x. long () <==> long(x)

lshift (...)  
x. lshift (y) <==> x<<y

neg (...)  
x. neg () <==> -x

nonzero (...)  
x. nonzero () <==> x != 0

oct (...)  
x. oct () <==> oct(x)

or (...)  
x. or (y) <==> x|y

pos (...)  
x. pos () <==> +x

pow (...)  
x. pow (y[, z]) <==> pow(x, y[, z])

rand (...)  
x. rand (y) <==> y&x

rdiv (...)  
x. rdiv (y) <==> y/x

rdivmod (...)  
x. rdivmod (y) <==> divmod(y, x)

reduce (...)  
a. reduce ()

    For pickling.

repr (...)  
x. repr () <==> repr(x)

rfloordiv (...)  
x. rfloordiv (y) <==> y//x

rlshift (...)  
x. rlshift (y) <==> y<<x

ror (...)  
x. ror (y) <==> y|x

rpow (...)  
y. rpow (x[, z]) <==> pow(x, y[, z])

rrshift (...)  
x. rrshift (y) <==> y>>x

rshift (...)  
x. rshift (y) <==> x>>y

rsub (...)  
x. rsub (y) <==> y-x

rtruediv (...)  
x. rtruediv (y) <==> y/x

rxor (...)  
x. rxor (y) <==> y^x

setitem (...)  
x. setitem (i, y) <==> x[i]=y

setslice (...)
```

```
x. __setslice__(i, j, y) <==> x[i:j]=y
Use of negative indices is not supported.

__setstate__(...)
a.__setstate__(version, shape, dtype, isfortran, rawdata)
For unpickling.

Parameters
-----
version : int
    optional pickle version. If omitted defaults to 0.
shape : tuple
dtype : data-type
isFortran : bool
rawdata : string or list
    a binary string with the data (or a list if 'a' is an object array)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
a.all(axis=None, out=None, keepdims=False)
Returns True if all elements evaluate to True.
Refer to `numpy.all` for full documentation.

See Also
-----
numpy.all : equivalent function

any(...)
a.any(axis=None, out=None, keepdims=False)
Returns True if any of the elements of `a` evaluate to True.
Refer to `numpy.any` for full documentation.

See Also
-----
numpy.any : equivalent function

argmax(...)
a.argmax(axis=None, out=None)
Return indices of the maximum values along the given axis.
Refer to `numpy.argmax` for full documentation.

See Also
-----
numpy.argmax : equivalent function

argmin(...)
a.argmin(axis=None, out=None)
Return indices of the minimum values along the given axis of `a`.
Refer to `numpy.argmin` for detailed documentation.

See Also
-----
numpy.argmin : equivalent function

argpartition(...)
a.argpartition(kth, axis=-1, kind='introselect', order=None)
Returns the indices that would partition this array.
Refer to `numpy.argpartition` for full documentation.
```

```
.. versionadded:: 1.8.0

See Also
-----
numpy.argpartition : equivalent function

astype(...)
a.astype\(dtype, order='K', casting='unsafe', subok=True, copy=True)

Copy of the array, cast to a specified type.

Parameters
-----
dtype : str or dtype
    Typecode or data-type to which the array is cast.
order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout order of the result.
    'C' means C order, 'F' means Fortran order, 'A'
    means 'F' order if all the arrays are Fortran contiguous,
    'C' order otherwise, and 'K' means as close to the
    order the array elements appear in memory as possible.
    Default is 'K'.
casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
    Controls what kind of data casting may occur. Defaults to 'unsafe'
    for backwards compatibility.

    * 'no' means the data types should not be cast at all.
    * 'equiv' means only byte-order changes are allowed.
    * 'safe' means only casts which can preserve values are allowed.
    * 'same_kind' means only safe casts or casts within a kind,
      like float64 to float32, are allowed.
    * 'unsafe' means any data conversions may be done.
subok : bool, optional
    If True, then sub-classes will be passed-through (default), otherwise
    the returned array will be forced to be a base-class array.
copy : bool, optional
    By default, astype always returns a newly allocated array. If this
    is set to false, and the `dtype`, `order`, and `subok`
    requirements are satisfied, the input array is returned instead
    of a copy.

Returns
-----
arr_t : ndarray
    Unless `copy` is False and the other conditions for returning the input
    array are satisfied (see description for `copy` input parameter), `arr_t`
    is a new array of the same shape as the input array, with dtype, order
    given by `dtype`, `order`.

Notes
-----
Starting in NumPy 1.9, astype method now returns an error if the string
dtype to cast to is not long enough in 'safe' casting mode to hold the max
value of integer/float array that is being casted. Previously the casting
was allowed even if the result was truncated.

Raises
-----
ComplexWarning
    When casting from complex to float or int. To avoid this,
    one should use ``a.real.astype(t)``.

Examples
-----
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1.,  2.,  2.5])

>>> x.astype(int)
array([1, 2, 2])

byteswap(...)
a.byteswap\(inplace\)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by
returning a byteswapped array, optionally swapped in-place.

Parameters
-----
inplace : bool, optional
    If ``True``, swap bytes in-place, default is ``False``.

Returns
-----
out : ndarray
```

The byteswapped array. If `inplace` is ``True``, this is a view to self.

Examples

```
----->>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256,       1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

choose(...)

```
a.choose(choices, out=None, mode='raise')
```

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also

```
-----numpy.choose : equivalent function
```

clip(...)

```
a.clip(min=None, max=None, out=None)
```

Return an array whose values are limited to ``[min, max]``. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See Also

```
-----numpy.clip : equivalent function
```

compress(...)

```
a.compress(condition, axis=None, out=None)
```

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

```
-----numpy.compress : equivalent function
```

conj(...)

```
a.conj()
```

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

```
-----numpy.conjugate : equivalent function
```

conjugate(...)

```
a.conjugate()
```

Return the [complex](#) conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See Also

```
-----numpy.conjugate : equivalent function
```

copy(...)

```
a.copy(order='C')
```

Return a copy of the array.

Parameters

```
-----order : {'C', 'F', 'A', 'K'}, optional
```

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if 'a' is Fortran contiguous,

'C' otherwise. 'K' means match the layout of `a` as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their order= arguments.)

See also

`numpy.copy`
`numpy.copyto`

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

cumprod(...)

`a.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also

`numpy.cumprod` : equivalent function

cumsum(...)

`a.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also

`numpy.cumsum` : equivalent function

diagonal(...)

`a.diagonal(offset=0, axis1=0, axis2=1)`

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to :func:`numpy.diagonal` for full documentation.

See Also

`numpy.diagonal` : equivalent function

dot(...)

`a.dot(b, out=None)`

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See Also

`numpy.dot` : equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
```

```
[ 8.,  8.]])

dump(...)  
    a.dump(file)  
  
        Dump a pickle of the array to the specified file.  
        The array can be read back with pickle.load or numpy.load.  
  
        Parameters  
        -----  
        file : str  
            A string naming the dump file.  
  
dumps(...)  
    a.dumps()  
  
        Returns the pickle of the array as a string.  
        pickle.loads or numpy.loads will convert the string back to an array.  
  
        Parameters  
        -----  
        None  
  
fill(...)  
    a.fill(value)  
  
        Fill the array with a scalar value.  
  
        Parameters  
        -----  
        value : scalar  
            All elements of `a` will be assigned this value.  
  
        Examples  
        -----  
        >>> a = np.array([1, 2])  
        >>> a.fill(0)  
        >>> a  
        array([0, 0])  
        >>> a = np.empty(2)  
        >>> a.fill(1)  
        >>> a  
        array([ 1.,  1.])  
  
flatten(...)  
    a.flatten(order='C')  
  
        Return a copy of the array collapsed into one dimension.  
  
        Parameters  
        -----  
        order : {'C', 'F', 'A', 'K'}, optional  
            'C' means to flatten in row-major (C-style) order.  
            'F' means to flatten in column-major (Fortran-style) order.  
            'A' means to flatten in column-major order if `a` is Fortran *contiguous* in memory,  
            row-major order otherwise. 'K' means to flatten `a` in the order the elements occur in memory.  
            The default is 'C'.  
  
        Returns  
        -----  
        y : ndarray  
            A copy of the input array, flattened to one dimension.  
  
        See Also  
        -----  
        ravel : Return a flattened array.  
        flat : A 1-D flat iterator over the array.  
  
        Examples  
        -----  
        >>> a = np.array([[1,2], [3,4]])  
        >>> a.flatten()  
        array([1, 2, 3, 4])  
        >>> a.flatten('F')  
        array([1, 3, 2, 4])  
  
getfield(...)  
    a.getfield(dtype, offset=0)  
  
        Returns a field of the given array as a certain type.  
  
        A field is a view of the array data with a given data-type. The values in  
        the view are determined by the given type and the offset into the current  
        array in bytes. The offset needs to be such that the view dtype fits in the
```

```

array dtype; for example an array of dtype complex128 has 16-byte elements.
If taking a view with a 32-bit integer (4 bytes), the offset needs to be
between 0 and 12 bytes.

Parameters
-----
dtype : str or dtype
    The data type of the view. The dtype size of the view can not be larger
    than that of the array itself.
offset : int
    Number of bytes to skip before beginning the element view.

Examples
-----
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
([[ 1.,  0.],
       [ 0.,  2.]])

By choosing an offset of 8 bytes we can select the complex part of the
array for our view:

>>> x.getfield(np.float64, offset=8)
([[ 1.,  0.],
       [ 0.,  4.]])

item(...)
a.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters
-----
**args : Arguments (variable number and type)

* none: in this case, the method only works for arrays
  with one element (`a.size == 1`), which element is
  copied into a standard Python scalar object and returned.

* int_type: this argument is interpreted as a flat index into
  the array, specifying which element to copy and return.

* tuple of int_types: functions as does a single int_type argument,
  except that the argument is interpreted as an nd-index into the
  array.

Returns
-----
z : Standard Python scalar object
    A copy of the specified element of the array as a suitable
    Python scalar

Notes
-----
When the data type of `a` is longdouble or clongdouble, item() returns
a scalar array object because there is no available Python scalar that
would not lose information. Void arrays return a buffer object for item(),
unless fields are defined, in which case a tuple is returned.

`item` is very similar to a[args], except, instead of an array scalar,
a standard Python scalar is returned. This can be useful for speeding up
access to elements of the array and doing arithmetic on elements of the
array using Python's optimized math.

Examples
-----
>>> x = np.random.randint(9, size=(3, 3))
>>> x
([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3

itemset(...)
a.itemset(*args)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

```

There must be at least 1 argument, and define the last argument as `*args`. Then, ```a.itemset(*args)``` is equivalent to but faster than ```a[args] = item```. The item should be a scalar value and `args` must select a [single](#) item in the array `'a'`.

Parameters

`*args : Arguments`

If one argument: a scalar, only used in case `'a'` is of size 1.
If two arguments: the last argument is the value to be set
and must be a scalar, the first argument specifies a [single](#) array element location. It is either an [int](#) or a tuple.

Notes

Compared to indexing syntax, `'itemset'` provides some speed increase for placing a scalar into a particular location in an `'ndarray'`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `'itemset'` (and `'item'`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max(...)
`a.max(axis=None, out=None)`

Return the maximum along a given axis.

Refer to `'numpy.amax'` for full documentation.

See Also

`numpyamax` : equivalent function

mean(...)
`a.mean(axis=None, dtype=None, out=None, keepdims=False)`

Returns the average of the array elements along given axis.

Refer to `'numpy.mean'` for full documentation.

See Also

`numpy.mean` : equivalent function

min(...)
`a.min(axis=None, out=None, keepdims=False)`

Return the minimum along a given axis.

Refer to `'numpy.amin'` for full documentation.

See Also

`numpy.amin` : equivalent function

newbyteorder(...)
`arr.newbyteorder(new_order='S')`

Return the array with the same data viewed with a different [byte](#) order.

Equivalent to::

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

```
new_order : string, optional
    Byte order to force; a value from the byte order specifications
    below. `new_order` codes can be any of:
        * 'S' - swap dtype from current to opposite endian
        * {'<', 'L'} - little endian
        * {'>', 'B'} - big endian
        * {'=', 'N'} - native order
        * {'|', 'I'} - ignore (no change to byte order)

    The default value ('S') results in swapping the current
    byte order. The code does a case-insensitive check on the first
    letter of `new_order` for the alternatives above. For example,
    any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

>Returns
-----
new_arr : array
    New array object with the dtype reflecting given change to the
    byte order.

nonzero(...)
a.nonzero\(\)

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See Also
-----
numpy.nonzero : equivalent function

prod(...)
a.prod(axis=None, dtype=None, out=None, keepdims=False)

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also
-----
numpy.prod : equivalent function

ptp(...)
a.ptp(axis=None, out=None)

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also
-----
numpy.ptp : equivalent function

put(...)
a.put(indices, values, mode='raise')

Set ``a.flat[n] = values[n]`` for all `n` in indices.

Refer to `numpy.put` for full documentation.

See Also
-----
numpy.put : equivalent function

ravel(...)
a.ravel([order])

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See Also
-----
numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat(...)
a.repeat(repeats, axis=None)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.
```

```
See Also
-----
numpy.repeat : equivalent function

reshape(...)
a.reshape(shape, order='C')

Returns an array containing the same data with a new shape.

Refer to 'numpy.reshape' for full documentation.

See Also
-----
numpy.reshape : equivalent function

resize(...)
a.resize(new_shape, refcheck=True)

Change shape and size of array in-place.

Parameters
-----
new_shape : tuple of ints, or `n` ints
    Shape of resized array.
refcheck : bool, optional
    If False, reference count will not be checked. Default is True.

Returns
-----
None

Raises
-----
ValueError
    If `a` does not own its own data or references or views to it exist,
    and the data memory must be changed.

SystemError
    If the 'order' keyword argument is specified. This behaviour is a
    bug in NumPy.

See Also
-----
resize : Return a new array with the specified shape.

Notes
-----
This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be
resized.

The purpose of the reference count check is to make sure you
do not use this array as a buffer for another Python object and then
reallocate the memory. However, reference counts can increase in
other ways so if you are sure that you have not shared the memory
for this array with another Python object, then you may safely set
`refcheck` to False.

Examples
-----
Shrinking an array: array is flattened (in the order that the data are
stored in memory), resized, and reshaped:

>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
```

```
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...

Unless `refcheck` is False:

>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])

round(...)
a.round(decimals=0, out=None)

Return `a` with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See Also
-----
numpy.around : equivalent function

searchsorted(...)
a.searchsorted(v, side='left', sorter=None)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted` 

See Also
-----
numpy.searchsorted : equivalent function

setfield(...)
a.setfield(val, dtype, offset=0)

Put a value into a specified place in a field defined by a data-type.

Place `val` into `a`'s field defined by `dtype` and beginning `offset` bytes into the field.

Parameters
-----
val : object
    Value to be placed in field.
dtype : dtype object
    Data-type of the field in which to place `val`.
offset : int, optional
    The number of bytes into the field at which to place `val`.

Returns
-----
None

See Also
-----
getfield

Examples
-----
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[ 1.0000000e+000,  1.48219694e-323,  1.48219694e-323],
       [ 1.48219694e-323,  1.0000000e+000,  1.48219694e-323],
       [ 1.48219694e-323,  1.48219694e-323,  1.0000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

```
setflags(...)
a.setflags(write=None, align=None, uic=None)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory
```

area used by `a` (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type.

The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

- write : bool, optional
 - Describes whether or not `a` can be written to.
- align : bool, optional
 - Describes whether or not `a` is aligned properly for its type.
- wic : bool, optional
 - Describes whether or not `a` is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
>>> y.setflags(wic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort(...)

a.sort(axis=-1, kind='quicksort', order=None)

Sort an array, in-place.

Parameters

- axis : [int](#), optional
 - Axis along which to sort. Default is -1, which means sort along the last axis.
- kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 - Sorting algorithm. Default is 'quicksort'.
- order : [str](#) or list of [str](#), optional
 - When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. A [single](#) field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the [dtype](#), to break ties.

See Also

- numpy.sort : Return a sorted copy of an array.
- argsort : Indirect sort.
- lexsort : Indirect stable sort on multiple keys.
- searchsorted : Find elements in sorted array.
- partition: Partial sort.

```
Notes
-----
See ``sort`` for notes on the different sorting algorithms.

Examples
-----
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])

Use the `order` keyword to specify a field to use when sorting a
structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])

squeeze(...)
a.squeeze(axis=None)

Remove single-dimensional entries from the shape of `a`.

Refer to `numpy.squeeze` for full documentation.

See Also
-----
numpy.squeeze : equivalent function

std(...)
a.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also
-----
numpy.std : equivalent function

sum(...)
a.sum(axis=None, dtype=None, out=None, keepdims=False)

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also
-----
numpy.sum : equivalent function

swapaxes(...)
a.swapaxes(axis1, axis2)

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also
-----
numpy.swapaxes : equivalent function

take(...)
a.take(indices, axis=None, out=None, mode='raise')

Return an array formed from the elements of `a` at the given indices.

Refer to `numpy.take` for full documentation.

See Also
-----
numpy.take : equivalent function

tobytes(...)
a.tobytes(order='C')

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of
```

```
data memory. The bytes object can be produced in either 'C' or 'Fortran',  
or 'Any' order (the default is 'C'-order). 'Any' order means C-order  
unless the F_CONTIGUOUS flag in the array is set, in which case it  
means 'Fortran' order.  
.. versionadded:: 1.9.0  
  
Parameters  
-----  
order : {'C', 'F', None}, optional  
    Order of the data for multidimensional arrays:  
    C, Fortran, or the same as for the original array.  
  
Returns  
-----  
s : bytes  
    Python bytes exhibiting a copy of `a`'s raw data.  
  
Examples  
-----  
>>> x = np.array([[0, 1], [2, 3]])  
>>> x.tobytes()  
b'\x00\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x03\x00\x00\x00'  
>>> x.tobytes('C') == x.tobytes()  
True  
>>> x.tobytes('F')  
b'\x00\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x03\x00\x00\x00'  
  
tofile(...)  
a.tofile(fid, sep="", format="%s")  
  
Write array to a file as text or binary (default).  
  
Data is always written in 'C' order, independent of the order of `a`.  
The data produced by this method can be recovered using the function  
fromfile\(\).  
  
Parameters  
-----  
fid : file or str  
    An open file object, or a string containing a filename.  
sep : str  
    Separator between array items for text output.  
    If "" (empty), a binary file is written, equivalent to  
    ``file.write(a.tobytes())``.  
format : str  
    Format string for text file output.  
    Each entry in the array is formatted to text by first converting  
    it to the closest Python type, and then using "format" % item.  
  
Notes  
-----  
This is a convenience function for quick storage of array data.  
Information on endianness and precision is lost, so this method is not a  
good choice for files intended to archive data or transport data between  
machines with different endianness. Some of these problems can be overcome  
by outputting the data as text files, at the expense of speed and file  
size.  
  
tolist(...)  
a.tolist()  
  
Return the array as a (possibly nested) list.  
  
Return a copy of the array data as a (nested) Python list.  
Data items are converted to the nearest compatible Python type.  
  
Parameters  
-----  
none  
  
Returns  
-----  
y : list  
    The possibly nested list of array elements.  
  
Notes  
-----  
The array may be recreated, ``a = np.array(a.tolist())``.  
  
Examples  
-----  
>>> a = np.array([1, 2])  
>>> a.tolist()  
[1, 2]  
>>> a = np.array([[1, 2], [3, 4]])  
>>> list(a)  
[[array([1, 2]), array([3, 4])]
```

```
>>> a.tolist\(\)
[[1, 2], [3, 4]]
```

tostring(...)
[a.tostring\(order='C'\)](#)

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes [object](#) can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the `F_CONTIGUOUS` flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for `tobytes`. Despite its name it returns bytes not strings.

Parameters

`order : {'C', 'F', None}, optional`
Order of the data for multidimensional arrays:
C, Fortran, or the same as for the original array.

Returns

`s : bytes`
Python bytes exhibiting a copy of `a`'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes\(\)
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes\('C'\) == x.tobytes\(\)
True
>>> x.tobytes\('F'\)
b'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

trace(...)
[a.trace\(offset=0, axis1=0, axis2=1, dtype=None, out=None\)](#)

Return the sum along diagonals of the array.

Refer to `'numpy.trace'` for full documentation.

See Also

[numpy.trace](#) : equivalent function

transpose(...)
[a.transpose\(*axes\)](#)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a [matrix object](#).)
For a 2-D array, this is the usual [matrix transpose](#).
For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then
```a.transpose\(\).shape = (i[n-1], i[n-2], ... i[1], i[0])```.

**Parameters**

-----

`axes : None, tuple of ints, or 'n' ints`

- \* None or no argument: reverses the order of the axes.
- \* tuple of ints: `i` in the `j`-th place in the tuple means `a`'s `i`-th axis becomes `a.[transpose\(\)](#)`'s `j`-th axis.
- \* `n` ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

-----

`out : ndarray`  
View of `a`, with axes suitably permuted.

**See Also**

-----

[ndarray.T](#) : Array property returning the array transposed.

**Examples**

-----

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
 [3, 4]])
```

```
>>> a.transpose()
array([[1, 3],
 [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
 [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
 [2, 4]])

var(...)
a.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

 Returns the variance of the array elements, along given axis.

 Refer to `numpy.var` for full documentation.

See Also

numpy.var : equivalent function

view(...)
a.view(dtype=None, type=None)

 New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional
 Data-type descriptor of the returned view, e.g., float32 or int16. The
 default, None, results in the view having the same data-type as `a`.
 This argument can also be specified as an ndarray sub-class, which
 then specifies the type of the returned object (this is equivalent to
 setting the ```type`` parameter).
type : Python type, optional
 Type of the returned view, e.g., ndarray or matrix. Again, the
 default None results in type preservation.

Notes

``a.view()`` is used two different ways:

``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a view
of the array's memory with a different data-type. This can cause a
reinterpretation of the bytes of memory.

``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` just
returns an instance of `ndarray_subclass` that looks at the same array
(same shape, dtype, etc.) This does not cause a reinterpretation of the
memory.

For ``a.view(some_dtype)``, if ``some_dtype`` has a different number of
bytes per entry than the previous dtype (for example, converting a
regular array to a structured array), then the behavior of the view
cannot be predicted just from the superficial appearance of ``a`` (shown
by ``print(a)``). It also depends on exactly how ``a`` is stored in
memory. Therefore if ``a`` is C-ordered versus fortran-ordered, versus
defined as a slice or transpose, etc., the view may give different
results.

Examples

>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
Viewing array data using a different type and dtype:
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>

Creating a view on a structured array so it can be used in calculations
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
 [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])

Making changes to the view changes the underlying array
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

```
Using a view to convert an array to a recarray:
>>> z = x.view\(np.recarray\)
>>> z.a
array\(\[1\], dtype=int8\)

Views share data:
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)

Views that change the dtype size (bytes per entry) should normally be
avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array\(\[\[1, 2\],
 \[4, 5\]\], dtype=int16\)
>>> y.view\(dtype=\[\('width', np.int16\), \('length', np.int16\)\]\)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy\(\)
>>> z.view\(dtype=\[\('width', np.int16\), \('length', np.int16\)\]\)
array\(\[\(1, 2\),
 \(4, 5\)\], dtype=\[\('width', <i2>\), \('length', <i2>\)\]\)
```

---

Data descriptors inherited from [numpy.ndarray](#):

## T

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

Examples

```
>>> x = np.array([[1., 2.], [3., 4.]])
>>> x
array([[1., 2.],
 [3., 4.]])
>>> x.T
array([[1., 3.],
 [2., 4.]])
>>> x = np.array([1., 2., 3., 4.])
>>> x
array([1., 2., 3., 4.])
>>> x.T
array([1., 2., 3., 4.])
```

## [array\\_interface](#)

Array protocol: Python side.

## [array\\_priority](#)

Array priority.

## [array\\_struct](#)

Array protocol: C-struct side.

## [base](#)

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

## [ctypes](#)

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used

```
as arguments to a shared library.

Parameters

None

Returns

c : Python object
 Possessing attributes data, shape, strides, etc.

See Also

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented
in "Guide to NumPy" (we have omitted undocumented public attributes,
as well as documented private attributes):

* data: A pointer to the memory area of the array as a Python integer.
 This memory area may contain data that is not aligned, or not in correct
 byte-order. The memory area may not even be writeable. The array
 flags and data-type of this array should be respected when passing this
 attribute to arbitrary C-code to avoid trouble that can include Python
 crashing. User Beware! The value of this attribute is exactly the same
 as self._array_interface_['data'][0].

* shape (c_intp*self.ndim): A ctypes array of length self.ndim where
 the basetype is the C-integer corresponding to dtype('p') on this
 platform. This base-type could be c_int, c_long, or c_longlong
 depending on the platform. The c_intp type is defined accordingly in
 numpy.ctypeslib. The ctypes array contains the shape of the underlying
 array.

* strides (c_intp*self.ndim): A ctypes array of length self.ndim where
 the basetype is the same as for the shape attribute. This ctypes array
 contains the strides information from the underlying array. This strides
 information is important for showing how many bytes must be jumped to
 get to the next element in the array.

* data_as(obj): Return the data pointer cast to a particular c-types object.
 For example, calling self._as_parameter_ is equivalent to
 self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a
 pointer to a ctypes array of floating-point data:
 self.data_as(ctypes.POINTER(ctypes.c_double)).

* shape_as(obj): Return the shape tuple as an array of some other c-types
 type. For example: self.shape_as(ctypes.c_short).

* strides_as(obj): Return the strides tuple as an array of some other
 c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary
arrays or arrays constructed on the fly. For example, calling
``(a+b).ctypes.data_as(ctypes.c_void_p)`` returns a pointer to memory
that is invalid because the array created as (a+b) is deallocated
before the next Python statement. You can avoid this problem using
either ``c=a+b`` or ``ct=(a+b).ctypes``. In the latter case, ct will
hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute
of array objects still returns something useful, but ctypes objects
are not returned and errors may be raised instead. In particular,
the object will still have the as parameter attribute which will
return an integer equal to the data attribute.

Examples

>>> import ctypes
>>> x
array([[0, 1],
 [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
```

```
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>

data
 Python buffer object pointing to the start of the array's data.

dtype
 Data-type of the array's elements.

 Parameters

 None

 Returns

 d : numpy dtype object

 See Also

 numpy.dtype

 Examples

 >>> x
 array([[0, 1],
 [2, 3]])
 >>> x.dtype
 dtype('int32')
 >>> type(x.dtype)
 <type 'numpy.dtype'>

flags
 Information about the memory layout of the array.

 Attributes

 C_CONTIGUOUS (C)
 The data is in a single, C-style contiguous segment.
 F_CONTIGUOUS (F)
 The data is in a single, Fortran-style contiguous segment.
 OWNDATA (O)
 The array owns the memory it uses or borrows it from another object.
 WRITEABLE (W)
 The data area can be written to. Setting this to False locks
 the data, making it read-only. A view (slice, etc.) inherits WRITEABLE
 from its base array at creation time, but a view of a writeable
 array may be subsequently locked while the base array remains writeable.
 (The opposite is not true, in that a view of a locked array may not
 be made writeable. However, currently, locking a base object does not
 lock any views that already reference it, so under that circumstance it
 is possible to alter the contents of a locked array via a previously
 created writeable view onto it.) Attempting to change a non-writeable
 array raises a RuntimeError exception.
 ALIGNED (A)
 The data and all elements are aligned appropriately for the hardware.
 UPDATEIFCOPY (U)
 This array is a copy of some other array. When this array is
 deallocated, the base array will be updated with the contents of
 this array.
 FNC
 F_CONTIGUOUS and not C_CONTIGUOUS.
 FORC
 F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
 BEHAVED (B)
 ALIGNED and WRITEABLE.
 CARRAY (CA)
 BEHAVED and C_CONTIGUOUS.
 FARRAY (FA)
 BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

 Notes

 The `flags` object can be accessed dictionary-like (as in ``a.flags['WRITEABLE']``),
 or by using lowercased attribute names (as in ``a.flags.writeable``). Short flag
 names are only supported in dictionary access.

 Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by
 the user, via direct assignment to the attribute or dictionary entry,
 or by calling `ndarray.setflags`.

 The array flags cannot be set arbitrarily:
 - UPDATEIFCOPY can only be set ``False``.
 - ALIGNED can only be set ``True`` if the data is truly aligned.
 - WRITEABLE can only be set ``True`` if the array owns its own memory
 or the ultimate owner of the memory exposes a writeable buffer
 interface or is a string.

 Arrays can be both C-style and Fortran-style contiguous simultaneously.
```

This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension ``arr.strides[dim]`` may be \*arbitrary\* if ``arr.shape[dim] == 1`` or the array has no elements. It does \*not\* generally hold that ``self.strides[-1] == self.itemsize`` for C-style contiguous arrays or ``self.strides[0] == self.itemsize`` for Fortran-style contiguous arrays is true.

## flat

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

-----  
flatten : Return a copy of the array collapsed into one dimension.

flatiter

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
 [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
 [2, 5],
 [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
 [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
 [3, 1, 3]])
```

## imag

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

## itemsize

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

## nbytes

Total bytes consumed by the elements of the array.

Notes

-----  
Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**ndim**

Number of array dimensions.

**Examples**

```

>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

**real**

The real part of the array.

**Examples**

```

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

**See Also**

`numpy.real` : equivalent function

**shape**

Tuple of array dimensions.

**Notes**

-----  
May be used to "reshape" the array, as long as this would not require a change in the total number of elements

**Examples**

```

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

**size**

Number of elements in the array.

Equivalent to ```np.prod(a.shape)```, i.e., the product of the array's dimensions.

**Examples**

```

>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ```(i[0], i[1], ..., i[n])``` in an array `a` is::

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

**Notes**

-----  
Imagine an array of 32-bit integers (each 4 bytes)::

```
x = np.array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell

us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be `(20, 4)`.

See Also

-----  
[numpy.lib.stride\\_tricks.as\\_strided](#)

Examples

```

>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]],
 [[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Data and other attributes inherited from [numpy.ndarray](#):

**\_\_hash\_\_** = None

**clongdouble** = class complex256([complexfloating](#))

Composed of two 128 bit floats

Method resolution order:

[complex256](#)  
[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[builtin](#) .[object](#)

Methods defined here:

**\_\_complex\_\_(...)**  
**\_\_eq\_\_(...)**  
 x. [\\_\\_eq\\_\\_](#)(y) <==> x==y  
**\_\_float\_\_(...)**  
 x. [\\_\\_float\\_\\_](#)() <==> [float](#)(x)  
**\_\_ge\_\_(...)**  
 x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y  
**\_\_gt\_\_(...)**  
 x. [\\_\\_gt\\_\\_](#)(y) <==> x>y  
**\_\_hash\_\_(...)**  
 x. [\\_\\_hash\\_\\_](#)() <==> hash(x)  
**\_\_hex\_\_(...)**  
 x. [\\_\\_hex\\_\\_](#)() <==> hex(x)  
**\_\_int\_\_(...)**

---

```

x.int() <==> int(x)

le(...)
x.le(y) <==> x<=y

long(...)
x.long() <==> long(x)

lt(...)
x.lt(y) <==> x<y

ne(...)
x.ne(y) <==> x!=y

oct(...)
x.oct() <==> oct(x)

repr(...)
x.repr() <==> repr(x)

str(...)
x.str() <==> str(x)

```

---

Data and other attributes defined here:

new = <built-in method new of type object>  
T.new(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from generic:

```

abs(...)
x.abs() <==> abs(x)

add(...)
x.add(y) <==> x+y

and(...)
x.and(y) <==> x&y

array(...)
sc.array(|type) return 0-dim array

array_wrap(...)
sc.array_wrap(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
x.div(y) <==> x/y

divmod(...)
x.divmod(y) <==> divmod(x, y)

floordiv(...)
x.floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x.getitem(y) <==> x[y]

invert(...)
x.invert() <==> ~x

lshift(...)
x.lshift(y) <==> x<<y

mod(...)
x.mod(y) <==> x%y

```

```
__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y
```

**\_\_xor\_\_(...)**  
x.xor(y) <=> x^y

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **conj(...)**

### **conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)****tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

```
__array_interface__
 Array protocol: Python side

__array_priority__
 Array priority.

__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

**clongfloat** = class complex256([complexfloating](#))  
Composed of two 128 bit floats

Method resolution order:  
[complex256](#)  
[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[\\_builtin\\_.object](#)

---

Methods defined here:

```
__complex__(...)
__eq__(...)
 x. eq(y) <==> x==y
__float__(...)
 x. float() <==> float(x)
__ge__(...)
```

---

```

x. __ge__(y) <==> x>=y

__gt__(...)
x. __gt__(y) <==> x>y

__hash__(...)
x. __hash__() <==> hash(x)

__hex__(...)
x. __hex__() <==> hex(x)

__int__(...)
x. __int__() <==> int(x)

__le__(...)
x. __le__(y) <==> x<=y

__long__(...)
x. __long__() <==> long(x)

__lt__(...)
x. __lt__(y) <==> x<y

__ne__(...)
x. __ne__(y) <==> x!=y

__oct__(...)
x. __oct__() <==> oct(x)

__repr__(...)
x. __repr__() <==> repr(x)

__str__(...)
x. __str__() <==> str(x)

```

---

Data and other attributes defined here:

---

```

__new__ = <built-in method __new__ of type object>
T. __new__(S, ...) -> a new object with type S, a subtype of T

```

---

Methods inherited from [generic](#):

---

```

__abs__(...)
x. __abs__() <==> abs(x)

__add__(...)
x. __add__(y) <==> x+y

__and__(...)
x. __and__(y) <==> x&y

__array__(...)
sc. __array__(|type) return 0-dim array

__array_wrap__(...)
sc. __array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x. __div__(y) <==> x/y

__divmod__(...)
x. __divmod__(y) <==> divmod(x, y)

__floordiv__(...)
x. __floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

```

```
__getitem__(...)
x.getitem(y) <==> x[y]

__invert__(...)
x.invert() <==> ~x

__lshift__(...)
x.lshift(y) <==> x<<y

__mod__(...)
x.mod(y) <==> x%y

__mul__(...)
x.mul(y) <==> x*y

__neg__(...)
x.neg() <==> -x

__nonzero__(...)
x.nonzero() <==> x != 0

__or__(...)
x.or(y) <==> x|y

__pos__(...)
x.pos() <==> +x

__pow__(...)
x.pow(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.radd(y) <==> y+x

__rand__(...)
x.rand(y) <==> y&x

__rdiv__(...)
x.rdiv(y) <==> y/x

__rdivmod__(...)
x.rdivmod(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.rfloordiv(y) <==> y//x

__rlshift__(...)
x.rlshift(y) <==> y<<x

__rmod__(...)
x.rmod(y) <==> y%x

__rmul__(...)
x.rmul(y) <==> y*x

__ror__(...)
x.ror(y) <==> y|x

__rpow__(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.rrshift(y) <==> y>>x

__rshift__(...)
x.rshift(y) <==> x>>y

__rsub__(...)
x.rsub(y) <==> y-x

__rtruediv__(...)
x.rtruediv(y) <==> y/x
```

```
__rxor__(...)
 x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
 x.__sub__(y) <==> x-y

__truediv__(...)
 x.__truediv__(y) <==> x/y

__xor__(...)
 x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

astype(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

toString(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

**\_\_array\_interface\_\_**

Array protocol: Python side

**\_\_array\_priority\_\_**

Array priority.

**\_\_array\_struct\_\_**

Array protocol: struct

**base**

base object

**data**

pointer to start of data

**dtype**

get array data-descriptor

**flags**

integer value of flags

**flat**

a 1-d view of scalar

**imag**

imaginary part of scalar

**itemsize**

length of one element in bytes

**nbytes**

length of item in bytes

**ndim**

number of array dimensions

**real**

real part of scalar

**shape**

tuple of array dimensions

**size**

number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

---

**class complex128(complexfloating, [builtin\\_.complex](#))**

Composed of two 64 bit floats

Method resolution order:

[complex128](#)

[complexfloating](#)

[inexact](#)

[number](#)

[generic](#)

[builtin\\_.complex](#)

[builtin\\_object](#)

Methods defined here:

[\\_\\_eq\\_\\_](#)(...)  
 x. [\\_\\_eq\\_\\_](#)(y) <==> x==y  
  
[\\_\\_ge\\_\\_](#)(...)  
 x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y  
  
[\\_\\_gt\\_\\_](#)(...)  
 x. [\\_\\_gt\\_\\_](#)(y) <==> x>y  
  
[\\_\\_le\\_\\_](#)(...)  
 x. [\\_\\_le\\_\\_](#)(y) <==> x<=y  
  
[\\_\\_lt\\_\\_](#)(...)  
 x. [\\_\\_lt\\_\\_](#)(y) <==> x<y  
  
[\\_\\_ne\\_\\_](#)(...)  
 x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y  
  
[\\_\\_repr\\_\\_](#)(...)  
 x. [\\_\\_repr\\_\\_](#)() <==> repr(x)  
  
[\\_\\_str\\_\\_](#)(...)  
 x. [\\_\\_str\\_\\_](#)() <==> [str](#)(x)

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
 T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

Methods inherited from [generic](#):

[\\_\\_abs\\_\\_](#)(...)  
 x. [\\_\\_abs\\_\\_](#)() <==> abs(x)  
  
[\\_\\_add\\_\\_](#)(...)  
 x. [\\_\\_add\\_\\_](#)(y) <==> x+y  
  
[\\_\\_and\\_\\_](#)(...)  
 x. [\\_\\_and\\_\\_](#)(y) <==> x&y  
  
[\\_\\_array\\_\\_](#)(...)  
 sc. [\\_\\_array\\_\\_](#)(|type) return 0-dim array  
  
[\\_\\_array\\_wrap\\_\\_](#)(...)  
 sc. [\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array  
  
[\\_\\_copy\\_\\_](#)(...)  
  
[\\_\\_deepcopy\\_\\_](#)(...)  
  
[\\_\\_div\\_\\_](#)(...)  
 x. [\\_\\_div\\_\\_](#)(y) <==> x/y  
  
[\\_\\_divmod\\_\\_](#)(...)  
 x. [\\_\\_divmod\\_\\_](#)(y) <==> divmod(x, y)  
  
[\\_\\_float\\_\\_](#)(...)  
 x. [\\_\\_float\\_\\_](#)() <==> [float](#)(x)  
  
[\\_\\_floordiv\\_\\_](#)(...)  
 x. [\\_\\_floordiv\\_\\_](#)(y) <==> x//y  
  
[\\_\\_format\\_\\_](#)(...)  
 NumPy array scalar formatter  
  
[\\_\\_getitem\\_\\_](#)(...)  
 x. [\\_\\_getitem\\_\\_](#)(y) <==> x[y]

```
hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

rmod(...)
x._rmod_(y) <==> y%x

rmul(...)
x._rmul_(y) <==> y*x

ror(...)
x._ror_(y) <==> y|x

rpow(...)
y._rpow_(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x._rrshift_(y) <==> y>>x
```

```
__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

astype(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

[new\\_order](#) : [str](#), optional

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

[new\\_dtype](#) : [dtype](#)

New `dtype` [object](#) with the given change to the [byte](#) order.

### **nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
swapaxes(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
tobytes(...)
tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
toString(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
var(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_\_array\_interface\_\_**  
Array protocol: Python side

**\_\_array\_priority\_\_**  
Array priority.

**\_\_array\_struct\_\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

Methods inherited from [builtin\\_.complex](#):

```
__coerce__(...)
 x. __coerce__(y) <==> coerce(x, y)

__getattribute__(...)
 x. __getattribute__('name') <==> x.name

__getnewargs__(...)

__hash__(...)
 x. __hash__() <==> hash(x)
```

---

**class complex256(complexfloating)**

Composed of two 128 bit floats

Method resolution order:

```
complex256
complexfloating
inexact
number
generic
__builtin__.object
```

---

Methods defined here:

```
__complex__(...)

__eq__(...)
 x. __eq__(y) <==> x==y

__float__(...)
 x. __float__() <==> float(x)

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__hash__(...)
 x. __hash__() <==> hash(x)

__hex__(...)
 x. __hex__() <==> hex(x)

__int__(...)
 x. __int__() <==> int(x)

__le__(...)
 x. __le__(y) <==> x<=y

__long__(...)
 x. __long__() <==> long(x)

__lt__(...)
 x. __lt__(y) <==> x<y

__ne__(...)
 x. __ne__(y) <==> x!=y

__oct__(...)
 x. __oct__() <==> oct(x)

__repr__(...)
 x. __repr__() <==> repr(x)

__str__(...)
 x. __str__() <==> str(x)
```

---

Data and other attributes defined here:

`_new_ = <built-in method __new__ of type object>`  
`T.new_(S, ...) -> a new object with type S, a subtype of T`

---

Methods inherited from [generic](#):

`_abs_(...)`  
`x.abs_() <==> abs(x)`

`_add_(...)`  
`x.add_(y) <==> x+y`

`_and_(...)`  
`x.and_(y) <==> x&y`

`_array_(...)`  
`sc.array_(|type) return 0-dim array`

`_array_wrap_(...)`  
`sc.array_wrap_(obj) return scalar from array`

`_copy_(...)`

`_deepcopy_(...)`

`_div_(...)`  
`x.div_(y) <==> x/y`

`_divmod_(...)`  
`x.divmod_(y) <==> divmod(x, y)`

`_floordiv_(...)`  
`x.floordiv_(y) <==> x//y`

`_format_(...)`  
`NumPy array scalar formatter`

`_getitem_(...)`  
`x.getitem_(y) <==> x[y]`

`_invert_(...)`  
`x.invert_() <==> ~x`

`_lshift_(...)`  
`x.lshift_(y) <==> x<<y`

`_mod_(...)`  
`x.mod_(y) <==> x%y`

`_mul_(...)`  
`x.mul_(y) <==> x*y`

`_neg_(...)`  
`x.neg_() <==> -x`

`_nonzero_(...)`  
`x.nonzero_() <==> x != 0`

`_or_(...)`  
`x.or_(y) <==> x|y`

`_pos_(...)`  
`x.pos_() <==> +x`

`_pow_(...)`  
`x.pow_(y[, z]) <==> pow(x, y[, z])`

`_radd_(...)`  
`x.radd_(y) <==> y+x`

`_rand_(...)`  
`x.rand_(y) <==> y&x`

`_rdiv_(...)`

```
x.__rdiv__(y) <==> y/x
__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)
__reduce__(...)
__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x
__rlshift__(...)
x.__rlshift__(y) <==> y<<x
__rmod__(...)
x.__rmod__(y) <==> y%x
__rmul__(...)
x.__rmul__(y) <==> y*x
__ror__(...)
x.__ror__(y) <==> y|x
__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])
__rrshift__(...)
x.__rrshift__(y) <==> y>>x
__rshift__(...)
x.__rshift__(y) <==> x>>y
__rsub__(...)
x.__rsub__(y) <==> y-x
__rtruediv__(...)
x.__rtruediv__(y) <==> y/x
__rxor__(...)
x.__rxor__(y) <==> y^x
__setstate__(...)
__sizeof__(...)
__sub__(...)
x.__sub__(y) <==> x-y
__truediv__(...)
x.__truediv__(y) <==> x/y
__xor__(...)
x.__xor__(y) <==> x^y
all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

```
newbyteorder(new_order='S')

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

repeat(...)

 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

**class complex64([complexfloating](#))**

Composed of two 32 bit floats

Method resolution order:

[complex64](#)  
[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[builtin\\_object](#)

---

Methods defined here:

[\\_\\_complex\\_\\_\(...\)](#)  
[\\_\\_eq\\_\\_\(...\)](#)  
    x. [\\_\\_eq\\_\\_\(y\)](#) <==> x==y  
[\\_\\_ge\\_\\_\(...\)](#)  
    x. [\\_\\_ge\\_\\_\(y\)](#) <==> x>=y  
[\\_\\_gt\\_\\_\(...\)](#)  
    x. [\\_\\_gt\\_\\_\(y\)](#) <==> x>y  
[\\_\\_hash\\_\\_\(...\)](#)  
    x. [\\_\\_hash\\_\\_\(\)](#) <==> hash(x)  
[\\_\\_le\\_\\_\(...\)](#)  
    x. [\\_\\_le\\_\\_\(y\)](#) <==> x<=y  
[\\_\\_lt\\_\\_\(...\)](#)  
    x. [\\_\\_lt\\_\\_\(y\)](#) <==> x<y  
[\\_\\_ne\\_\\_\(...\)](#)  
    x. [\\_\\_ne\\_\\_\(y\)](#) <==> x!=y  
[\\_\\_repr\\_\\_\(...\)](#)  
    x. [\\_\\_repr\\_\\_\(\)](#) <==> repr(x)  
[\\_\\_str\\_\\_\(...\)](#)  
    x. [\\_\\_str\\_\\_\(\)](#) <==> str(x)

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method \_\_new\_\_ of type object>  
    T. [\\_\\_new\\_\\_\(S, ...\)](#) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [generic](#):

[abs](#)(...)  
x. [abs](#)() <==> abs(x)

[add](#)(...)  
x. [add](#)(y) <==> x+y

[and](#)(...)  
x. [and](#)(y) <==> x&y

[array](#)(...)  
sc. [array](#)(|type) return 0-dim array

[array\\_wrap](#)(...)  
sc. [array\\_wrap](#)(obj) return scalar from array

[copy](#)(...)

[deepcopy](#)(...)

[div](#)(...)  
x. [div](#)(y) <==> x/y

[divmod](#)(...)  
x. [divmod](#)(y) <==> divmod(x, y)

[float](#)(...)  
x. [float](#)() <==> float(x)

[floordiv](#)(...)  
x. [floordiv](#)(y) <==> x//y

[format](#)(...)  
NumPy array scalar formatter

[getitem](#)(...)  
x. [getitem](#)(y) <==> x[y]

[hex](#)(...)  
x. [hex](#)() <==> hex(x)

[int](#)(...)  
x. [int](#)() <==> int(x)

[invert](#)(...)  
x. [invert](#)() <==> ~x

[long](#)(...)  
x. [long](#)() <==> long(x)

[lshift](#)(...)  
x. [lshift](#)(y) <==> x<<y

[mod](#)(...)  
x. [mod](#)(y) <==> x%y

[mul](#)(...)  
x. [mul](#)(y) <==> x\*y

[neg](#)(...)  
x. [neg](#)() <==> -x

[nonzero](#)(...)  
x. [nonzero](#)() <==> x != 0

[oct](#)(...)  
x. [oct](#)() <==> oct(x)

[or](#)(...)  
x. [or](#)(y) <==> x|y

[pos](#)(...)

```
x. __pos__() <==> +x

__pow__(...)
x. __pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x. __radd__(y) <==> y+x

__rand__(...)
x. __rand__(y) <==> y&x

__rdiv__(...)
x. __rdiv__(y) <==> y/x

__rdivmod__(...)
x. __rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
x. __rlshift__(y) <==> y<<x

__rmod__(...)
x. __rmod__(y) <==> y%x

__rmul__(...)
x. __rmul__(y) <==> y*x

__ror__(...)
x. __ror__(y) <==> y|x

__rpow__(...)
y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x. __rrshift__(y) <==> y>>x

__rshift__(...)
x. __rshift__(y) <==> x>>y

__rsub__(...)
x. __rsub__(y) <==> y-x

__rtruediv__(...)
x. __rtruediv__(y) <==> y/x

__rxor__(...)
x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
x. __sub__(y) <==> x-y

__truediv__(...)
x. __truediv__(y) <==> x/y

__xor__(...)
x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmax(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
compress(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)
conjugate(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dumps(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

```
min(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder\(new_order='S'\)

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 Parameters

 new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

**complex\_** = class complex128([complexfloating](#), [\\_\\_builtin\\_\\_.complex](#))  
Composed of two 64 bit floats

Method resolution order:

[complex128](#)  
[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[\\_\\_builtin\\_\\_.complex](#)  
[\\_\\_builtin\\_\\_.object](#)

---

Methods defined here:

[\*\*\\_\\_eq\\_\\_\(...\)\*\*](#)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\*\*\\_\\_ge\\_\\_\(...\)\*\*](#)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\*\*\\_\\_gt\\_\\_\(...\)\*\*](#)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\*\*\\_\\_le\\_\\_\(...\)\*\*](#)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\*\*\\_\\_lt\\_\\_\(...\)\*\*](#)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\*\*\\_\\_ne\\_\\_\(...\)\*\*](#)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

[\*\*\\_\\_repr\\_\\_\(...\)\*\*](#)  
x. [\\_\\_repr\\_\\_](#)() <==> repr(x)

**\_\_str\_\_**(...)  
x.\_\_str\_\_() <==> str(x)

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from generic:

**\_\_abs\_\_**(...)  
x.\_\_abs\_\_() <==> abs(x)  
  
**\_\_add\_\_**(...)  
x.\_\_add\_\_(y) <==> x+y  
  
**\_\_and\_\_**(...)  
x.\_\_and\_\_(y) <==> x&y  
  
**\_\_array\_\_**(...)  
sc.\_\_array\_\_(|type) return 0-dim array  
  
**\_\_array\_wrap\_\_**(...)  
sc.\_\_array\_wrap\_\_(obj) return scalar from array  
  
**\_\_copy\_\_**(...)  
  
**\_\_deepcopy\_\_**(...)  
  
**\_\_div\_\_**(...)  
x.\_\_div\_\_(y) <==> x/y  
  
**\_\_divmod\_\_**(...)  
x.\_\_divmod\_\_(y) <==> divmod(x, y)  
  
**\_\_float\_\_**(...)  
x.\_\_float\_\_() <==> float(x)  
  
**\_\_floordiv\_\_**(...)  
x.\_\_floordiv\_\_(y) <==> x//y  
  
**\_\_format\_\_**(...)  
NumPy array scalar formatter  
  
**\_\_getitem\_\_**(...)  
x.\_\_getitem\_\_(y) <==> x[y]  
  
**\_\_hex\_\_**(...)  
x.\_\_hex\_\_() <==> hex(x)  
  
**\_\_int\_\_**(...)  
x.\_\_int\_\_() <==> int(x)  
  
**\_\_invert\_\_**(...)  
x.\_\_invert\_\_() <==> ~x  
  
**\_\_long\_\_**(...)  
x.\_\_long\_\_() <==> long(x)  
  
**\_\_lshift\_\_**(...)  
x.\_\_lshift\_\_(y) <==> x<<y  
  
**\_\_mod\_\_**(...)  
x.\_\_mod\_\_(y) <==> x%y  
  
**\_\_mul\_\_**(...)  
x.\_\_mul\_\_(y) <==> x\*y  
  
**\_\_neg\_\_**(...)  
x.\_\_neg\_\_() <==> -x  
  
**\_\_nonzero\_\_**(...)  
x.\_\_nonzero\_\_() <==> x != 0

```
oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

sub(...)
x.sub(y) <==> x-y

truediv(...)
x.truediv(y) <==> x/y

xor(...)
x.xor(y) <==> x^y

all(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

---

**\_\_array\_struct\_\_**  
 Array protocol: struct

**base**  
 base object

**data**  
 pointer to start of data

**dtype**  
 get array data-descriptor

**flags**  
 integer value of flags

**flat**  
 a 1-d view of scalar

**imag**  
 imaginary part of scalar

**itemsize**  
 length of one element in bytes

**nbytes**  
 length of item in bytes

**ndim**  
 number of array dimensions

**real**  
 real part of scalar

**shape**  
 tuple of array dimensions

**size**  
 number of elements in the gentype

**strides**  
 tuple of bytes steps in each dimension

---

Methods inherited from [\\_\\_builtin\\_\\_.complex](#):

**\_\_coerce\_\_(...)**  
 $x.\underline{\text{coerce}}(y) \iff \text{coerce}(x, y)$

**\_\_getattribute\_\_(...)**  
 $x.\underline{\text{getattribute}}('name') \iff x.name$

**\_\_getnewargs\_\_(...)**

**\_\_hash\_\_(...)**  
 $x.\underline{\text{hash}}() \iff \text{hash}(x)$

---

class **complexfloating**([inexact](#))

Method resolution order:

[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[\\_\\_builtin\\_\\_.object](#)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
 $x.\underline{\text{abs}}() \iff \text{abs}(x)$

**\_\_add\_\_(...)**  
 $x.\underline{\text{add}}(y) \iff x+y$

```
__and__(...)
 x.__and__(y) <==> x&y

__array__(...)
 sc.__array__(|type) return 0-dim array

__array_wrap__(...)
 sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x.__div__(y) <==> x/y

__divmod__(...)
 x.__divmod__(y) <==> divmod(x, y)

__eq__(...)
 x.__eq__(y) <==> x==y

__float__(...)
 x.__float__() <==> float(x)

__floordiv__(...)
 x.__floordiv__(y) <==> x//y

__format__(...)
 NumPy array scalar formatter

__ge__(...)
 x.__ge__(y) <==> x>=y

__getitem__(...)
 x.__getitem__(y) <==> x[y]

__gt__(...)
 x.__gt__(y) <==> x>y

__hex__(...)
 x.__hex__() <==> hex(x)

__int__(...)
 x.__int__() <==> int(x)

__invert__(...)
 x.__invert__() <==> ~x

__le__(...)
 x.__le__(y) <==> x<=y

__long__(...)
 x.__long__() <==> long(x)

__lshift__(...)
 x.__lshift__(y) <==> x<<y

__lt__(...)
 x.__lt__(y) <==> x<y

__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(y) <==> x*y

__ne__(...)
 x.__ne__(y) <==> x!=y

__neg__(...)
 x.__neg__() <==> -x

__nonzero__(...)
```

```
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y
```

```
truediv(...)
x._truediv_(y) <==> x/y

xor(...)
x._xor_(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New [dtype](#) object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
tobytes(...)
tofile(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tolist(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

toString(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

**csingle** = class complex64([complexfloating](#))

Composed of two 32 bit floats

Method resolution order:

[complex64](#)  
[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[\\_builtin\\_.object](#)

---

Methods defined here:[\\_\\_complex\\_\\_\(...\)](#)  
[\\_\\_eq\\_\\_\(...\)](#)  
    x. [\\_\\_eq\\_\\_\(y\)](#) <==> x==y  
[\\_\\_ge\\_\\_\(...\)](#)

---

```

x. __ge__(y) <==> x>=y

__gt__(...)
x. __gt__(y) <==> x>y

__hash__(...)
x. __hash__() <==> hash(x)

__le__(...)
x. __le__(y) <==> x<=y

__lt__(...)
x. __lt__(y) <==> x<y

__ne__(...)
x. __ne__(y) <==> x!=y

__repr__(...)
x. __repr__() <==> repr(x)

__str__(...)
x. __str__() <==> str(x)

```

---

Data and other attributes defined here:

---

```

__new__ = <built-in method __new__ of type object>
T. __new__(S, ...) -> a new object with type S, a subtype of T

```

---

Methods inherited from [generic](#):

```

__abs__(...)
x. __abs__() <==> abs(x)

__add__(...)
x. __add__(y) <==> x+y

__and__(...)
x. __and__(y) <==> x&y

__array__(...)
sc. __array__(|type) return 0-dim array

__array_wrap__(...)
sc. __array_wrap__(obj) return scalar from array

__copy__()

__deepcopy__(...)

__div__(...)
x. __div__(y) <==> x/y

__divmod__(...)
x. __divmod__(y) <==> divmod(x, y)

__float__(...)
x. __float__() <==> float(x)

__floordiv__(...)
x. __floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x. __getitem__(y) <==> x[y]

__hex__(...)
x. __hex__() <==> hex(x)

__int__(...)
x. __int__() <==> int(x)

```

```
__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__() (y) <==> x<<y

__mod__(...)
x.__mod__() (y) <==> x%y

__mul__(...)
x.__mul__() (y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__() (y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__() (y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__() (y) <==> y+x

__rand__(...)
x.__rand__() (y) <==> y&x

__rdiv__(...)
x.__rdiv__() (y) <==> y/x

__rdivmod__(...)
x.__rdivmod__() (y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.__rfloordiv__() (y) <==> y//x

__rlshift__(...)
x.__rlshift__() (y) <==> y<<x

__rmod__(...)
x.__rmod__() (y) <==> y%x

__rmul__(...)
x.__rmul__() (y) <==> y*x

__ror__(...)
x.__ror__() (y) <==> y|x

__rpow__(...)
y.__rpow__() (x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__() (y) <==> y>>x

__rshift__(...)
x.__rshift__() (y) <==> x>>y

__rsub__(...)
x.__rsub__() (y) <==> y-x
```

```
__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

astype(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also
```

```

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

choose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----

new\_order : [str](#), optional  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----

new\_dtype : [dtype](#)  
New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **tobytes(...)**

#### **tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

**\_array\_interface\_**

Array protocol: Python side

**\_array\_priority\_**

Array priority.

**\_array\_struct\_**

Array protocol: struct

**base**

base object

**data**

pointer to start of data

**dtype**

get array data-descriptor

**flags**

integer value of flags

**flat**

a 1-d view of scalar

**imag**

imaginary part of scalar

**itemsize**

length of one element in bytes

**nbytes**

length of item in bytes

**ndim**

number of array dimensions

**real**

real part of scalar

**shape**

tuple of array dimensions

**size**

number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

class **datetime64**([generic](#))

Method resolution order:

[datetime64](#)  
[generic](#)  
[builtin\\_object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_](#)(...)  
x.[\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_ge\\_\\_](#)(...)  
x.[\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_](#)(...)  
x.[\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_](#)(...)  
x.[\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_le\\_\\_](#)(...)  
x.[\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_lt\\_\\_](#)(...)  
x.[\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_](#)(...)  
x.[\\_\\_ne\\_\\_](#)(y) <==> x!=y

[\\_\\_repr\\_\\_](#)(...)  
x.[\\_\\_repr\\_\\_](#)() <==> repr(x)

[\\_\\_str\\_\\_](#)(...)  
x.[\\_\\_str\\_\\_](#)() <==> str(x)

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method \_\_new\_\_ of type object>  
T.[\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [generic](#):

[\\_\\_abs\\_\\_](#)(...)  
x.[\\_\\_abs\\_\\_](#)() <==> abs(x)

[\\_\\_add\\_\\_](#)(...)  
x.[\\_\\_add\\_\\_](#)(y) <==> x+y

[\\_\\_and\\_\\_](#)(...)  
x.[\\_\\_and\\_\\_](#)(y) <==> x&y

[\\_\\_array\\_\\_](#)(...)  
sc.[\\_\\_array\\_\\_](#)(|type) return 0-dim array

[\\_\\_array\\_wrap\\_\\_](#)(...)  
sc.[\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array

[\\_\\_copy\\_\\_](#)(...)

[\\_\\_deepcopy\\_\\_](#)(...)

[\\_\\_div\\_\\_](#)(...)  
x.[\\_\\_div\\_\\_](#)(y) <==> x/y

[\\_\\_divmod\\_\\_](#)(...)  
x.[\\_\\_divmod\\_\\_](#)(y) <==> divmod(x, y)

[\\_\\_float\\_\\_](#)(...)  
x.[\\_\\_float\\_\\_](#)() <==> float(x)

[\\_\\_floordiv\\_\\_](#)(...)  
x.[\\_\\_floordiv\\_\\_](#)(y) <==> x//y

```
format(...)
 NumPy array scalar formatter

getitem(...)
 x. getitem (y) <==> x[y]

hex(...)
 x. hex () <==> hex(x)

int(...)
 x. int () <==> int(x)

invert(...)
 x. invert () <==> ~x

long(...)
 x. long () <==> long(x)

lshift(...)
 x. lshift (y) <==> x<<y

mod(...)
 x. mod (y) <==> x%y

mul(...)
 x. mul (y) <==> x*y

neg(...)
 x. neg () <==> -x

nonzero(...)
 x. nonzero () <==> x != 0

oct(...)
 x. oct () <==> oct(x)

or(...)
 x. or (y) <==> x|y

pos(...)
 x. pos () <==> +x

pow(...)
 x. pow (y[, z]) <==> pow(x, y[, z])

radd(...)
 x. radd (y) <==> y+x

rand(...)
 x. rand (y) <==> y&x

rdiv(...)
 x. rdiv (y) <==> y/x

rdivmod(...)
 x. rdivmod (y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
 x. rfloordiv (y) <==> y//x

rlshift(...)
 x. rlshift (y) <==> y<<x

rmod(...)
 x. rmod (y) <==> y%x

rmul(...)
 x. rmul (y) <==> y*x

ror(...)
 x. ror (y) <==> y|x
```

```
__rpow__(...)
 y. __rpow__ (x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x. __rrshift__ (y) <==> y>>x

__rshift__(...)
 x. __rshift__ (y) <==> x>>y

__rsub__(...)
 x. __rsub__ (y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__ (y) <==> y/x

__rxor__(...)
 x. __rxor__ (y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
 x. __sub__ (y) <==> x-y

__truediv__(...)
 x. __truediv__ (y) <==> x/y

__xor__(...)
 x. __xor__ (y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
```

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_\_array\_interface\_\_**  
Array protocol: Python side

**\_\_array\_priority\_\_**  
Array priority.

**\_\_array\_struct\_\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

---

**double** = class float64([floating](#), [builtin .float](#))  
 64-bit [floating-point number](#). Character code 'd'. Python [float](#) compatible.

Method resolution order:

```

float64
floating
inexact
number
generic
builtin .float
builtin .object

```

---

Methods defined here:

```

__eq__(...)
 x. __eq__(y) <==> x==y

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__le__(...)
 x. __le__(y) <==> x<=y

__lt__(...)
 x. __lt__(y) <==> x<y

__ne__(...)
 x. __ne__(y) <==> x!=y

__repr__(...)
 x. __repr__() <==> repr(x)

__str__(...)
 x. __str__() <==> str(x)

```

---

Data and other attributes defined here:

```

__new__ = <built-in method __new__ of type object>
 T. __new__(S, ...) -> a new object with type S, a subtype of T

```

---

Methods inherited from [generic](#):

```

__abs__(...)
 x. __abs__() <==> abs(x)

__add__(...)
 x. __add__(y) <==> x+y

__and__(...)
 x. __and__(y) <==> x&y

__array__(...)
 sc. __array__(|type) return 0-dim array

__array_wrap__(...)
 sc. __array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x. __div__(y) <==> x/y

```

```
divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x
```

**\_\_rmod\_\_(...)**  
x. rmod(y) <==> y%x

**\_\_rmul\_\_(...)**  
x. rmul(y) <==> y\*x

**\_\_ror\_\_(...)**  
x. ror(y) <==> y|x

**\_\_rpow\_\_(...)**  
y. rpow(x[, z]) <==> pow(x, y[, z])

**\_\_rrshift\_\_(...)**  
x. rrshift(y) <==> y>>x

**\_\_rshift\_\_(...)**  
x. rshift(y) <==> x>>y

**\_\_rsub\_\_(...)**  
x. rsub(y) <==> y-x

**\_\_rtruediv\_\_(...)**  
x. rtruediv(y) <==> y/x

**\_\_rxor\_\_(...)**  
x. rxor(y) <==> y^x

**\_\_setstate\_\_(...)**

**\_\_sizeof\_\_(...)**

**\_\_sub\_\_(...)**  
x. sub(y) <==> x-y

**\_\_truediv\_\_(...)**  
x. truediv(y) <==> x/y

**\_\_xor\_\_(...)**  
x. xor(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

```

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New 'dtype' object with the given change to the byte order.

nonzero\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
 real part of scalar

**shape**  
 tuple of array dimensions

**size**  
 number of elements in the gentype

**strides**  
 tuple of bytes steps in each dimension

---

Methods inherited from [\\_builtin\\_.float](#):

**\_coerce\_(...)**  
`x. coerce(y) <==> coerce(x, y)`

**\_getattribute\_(...)**  
`x. getattribute('name') <==> x.name`

**\_getformat\_(...)**  
`float. getformat(typestr) -> string`  
 You probably don't want to use this function. It exists mainly to be used in Python's test suite.  
 typestr must be '[double](#)' or '[float](#)'. This function returns whichever of 'unknown', 'IEEE, big-endian' or 'IEEE, little-endian' best describes the format of [floating](#) point numbers used by the C type named by typestr.

**\_getnewargs\_(...)**

**\_hash\_(...)**  
`x. hash() <==> hash(x)`

**\_setformat\_(...)**  
`float. setformat(typestr, fmt) -> None`  
 You probably don't want to use this function. It exists mainly to be used in Python's test suite.  
 typestr must be '[double](#)' or '[float](#)'. fmt must be one of 'unknown', 'IEEE, big-endian' or 'IEEE, little-endian', and in addition can only be one of the latter two if it appears to match the underlying C reality.  
 Override the automatic determination of C-level [floating](#) point type.  
 This affects how floats are converted to and from binary strings.

**\_trunc\_(...)**  
 Return the Integral closest to x between 0 and x.

**as\_integer\_ratio(...)**  
`float.as\_integer\_ratio() -> (int, int)`  
 Return a pair of integers, whose ratio is exactly equal to the original [float](#) and with a positive denominator.  
 Raise OverflowError on infinities and a ValueError on NaNs.  
`>>> (10.0).as\_integer\_ratio()  
(10, 1)  
>>> (0.0).as\_integer\_ratio()  
(0, 1)  
>>> (-.25).as\_integer\_ratio()  
(-1, 4)`

**fromhex(...)**  
`float.fromhex(string) -> float`  
 Create a [floating](#)-point [number](#) from a hexadecimal string.  
`>>> float.fromhex('0x1.ffffp10')  
2047.984375  
>>> float.fromhex('-.0x1p-1074')  
-4.9406564584124654e-324`

**hex(...)**  
`float.hex() -> string`  
 Return a hexadecimal representation of a [floating](#)-point [number](#).  
`>>> (-0.1).hex()  
'-0x1.9999999999999ap-4'`

```

>>> 3.14159.hex()
'0x1.921f9f01b866ep+1'

is_integer(...)
 Return True if the float is an integer.
```

**class [dtype](#)([builtin .object](#))**

[dtype](#)(obj, align=False, copy=False)

Create a data type [object](#).

A numpy array is homogeneous, and contains elements described by a [dtype object](#). A [dtype object](#) can be constructed from different combinations of fundamental numeric types.

Parameters

-----

obj

Object to be converted to a data type [object](#).

align : bool, optional

Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be ``True`` only if `obj` is a dictionary or a comma-separated string. If a struct [dtype](#) is being created, this also sets a sticky alignment flag ``isalignedstruct``.

copy : bool, optional

Make a new copy of the data-type [object](#). If ``False``, the result may just be a reference to a built-in data-type [object](#).

See also

-----

[result\\_type](#)

Examples

-----

Using array-scalar type:

```
>>> np.dtype(np.int16)
dtype('int16')
```

Structured type, one field name 'f1', containing [int16](#):

```
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Structured type, one field named 'f1', in itself containing a structured type with one field:

```
>>> np.dtype([('f1', [('f1', np.int16)])])
dtype([('f1', [('f1', '<i2')])])
```

Structured type, two fields: the first field contains an unsigned [int](#), the second an [int32](#):

```
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```

Using array-protocol type strings:

```
>>> np.dtype([('a','f8'),('b','S10')])
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```
>>> np.dtype("14, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

Using tuples. ``[int](#)`` is a fixed type, 3 is the field's shape. ``[void](#)`` is a [flexible](#) type, here of size 10:

```
>>> np.dtype([('hello',(np.int,3)),('world',np.void,10)])
dtype([('hello', '<i4', 3), ('world', '|V10')])
```

Subdivide ``[int16](#)`` into 2 ``[int8](#)``'s, called x and y. 0 and 1 are the offsets in bytes:

```
>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype([('i2', [('x', '|i1'), ('y', '|i1')])])
```

Using dictionaries. Two fields named 'gender' and 'age':

```
>>> np.dtype({'names':['gender','age'], 'formats':[('S1',np.uint8)]})
dtype([('gender', 'S1'), ('age', '|u1')])
```

Offsets in bytes, here 0 and 25:

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

Methods defined here:

**`__eq__(...)`**  
`x.__eq__(y) <=> x==y`

**`__ge__(...)`**  
`x.__ge__(y) <=> x>=y`

**`__getitem__(...)`**  
`x.__getitem__(y) <=> x[y]`

**`__gt__(...)`**  
`x.__gt__(y) <=> x>y`

**`__hash__(...)`**  
`x.__hash__() <=> hash(x)`

**`__le__(...)`**  
`x.__le__(y) <=> x<=y`

**`__len__(...)`**  
`x.__len__() <=> len(x)`

**`__lt__(...)`**  
`x.__lt__(y) <=> x<y`

**`__mul__(...)`**  
`x.__mul__(n) <=> x*n`

**`__ne__(...)`**  
`x.__ne__(y) <=> x!=y`

**`__reduce__(...)`**

**`__repr__(...)`**  
`x.__repr__() <=> repr(x)`

**`__rmul__(...)`**  
`x.__rmul__(n) <=> n*x`

**`__setstate__(...)`**

**`__str__(...)`**  
`x.__str__() <=> str(x)`

**`newbyteorder(...)`**  
`newbyteorder(new_order='S')`

Return a new `dtype` with a different `byte` order.

Changes are also made in all fields and sub-arrays of the data type.

Parameters

-----

`new_order : string, optional`

Byte order to force; a value from the `byte` order specifications below. The default value ('S') results in swapping the current `byte` order. 'new\_order' codes can be any of:

- \* 'S' - swap `dtype` from current to opposite endian
- \* {'<', '>'} - little endian
- \* {'>', '<'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to `byte` order)

The code does a case-insensitive check on the first letter of 'new\_order' for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New `dtype` object with the given change to the `byte` order.

Notes

-----

Changes are also made in all fields and sub-arrays of the data type.

```

Examples

>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('||')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True

```

---

Data descriptors defined here:

### **alignment**

The required alignment (bytes) of this data-type according to the compiler.

More information is available in the C-API section of the manual.

### **base**

### **byteorder**

A character indicating the byte-order of this data-type object.

One of:

```

=====
'=' native
'<' little-endian
'>' big-endian
'|' not applicable
=====

```

All built-in data-type objects have byteorder either '=' or '|'.

Examples

```

>>> dt = np.dtype('i2')
>>> dt.byteorder
'='
>>> # endian is not relevant for 8 bit numbers
>>> np.dtype('i1').byteorder
'|'
>>> # or ASCII strings
>>> np.dtype('S2').byteorder
'|'
>>> # Even if specific code is given, and it is native
>>> # '=' is the byteorder
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> dt = np.dtype(native_code + 'i2')
>>> dt.byteorder
'='
>>> # Swapped code shows up as itself
>>> dt = np.dtype(swapped_code + 'i2')
>>> dt.byteorder == swapped_code
True

```

### **char**

A unique character code for each of the 21 different built-in types.

### **descr**

Array-interface compliant full description of the data-type.

The format is that required by the 'descr' key in the `\_\_array\_interface\_\_` attribute.

**fields**

Dictionary of named fields defined for this data type, or ``None``.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field::

```
(dtype, offset[, title])
```

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the ``ndarray.getfield`` and ``ndarray.setfield`` methods.

See Also

-----  
ndarray.getfield, ndarray.setfield

Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print(dt.fields)
{'grades': (dtype('float64',(2,)), 16), 'name': (dtype('|S16'), 0)}
```

**flags**

Bit-flags describing how this data type is to be interpreted.

Bit-masks are in `numpy.core.multiarray` as the constants `ITEM\_HASOBJECT`, `LIST\_PICKLE`, `ITEM\_IS\_POINTER`, `NEEDS\_INIT`, `NEEDS\_PYAPI`, `USE\_GETITEM`, `USE\_SETITEM`. A full explanation of these flags is in C-API documentation; they are largely useful for user-defined data-types.

**hasobject**

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won't.

**isalignedstruct**

Boolean indicating whether the dtype is a struct which maintains field alignment. This flag is sticky, so when combining multiple structs together, it is preserved and produces new dtypes which are also aligned.

**isbuiltin**

Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

```
= =====
0 if this is a structured array type, with fields
1 if this is a dtype compiled into numpy (such as ints, floats etc)
2 if the dtype is for a user-defined numpy type
 A user-defined type uses the numpy C-API machinery to extend
 numpy to handle a new array type. See
 :ref:`user.user-defined-data-types` in the Numpy manual.
= =====
```

Examples

```
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

**isnative**

Boolean indicating whether the byte order of this dtype is native to the platform.

**itemsize**

The element size of this data-type object.

For 18 of the 21 types this number is fixed by the data-type.  
 For the flexible data-types, this number can be anything.

**kind**

A character code (one of 'biufcmMOSUV') identifying the general kind of data.

```
= =====
b boolean
i signed integer
u unsigned integer
f floating-point
c complex floating-point
m timedelta
M datetime
O object
S (byte-)string
U Unicode
V void
= =====
```

**metadata****name**

A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.

**names**

Ordered list of field names, or ``None`` if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

Examples

```

>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.names
('name', 'grades')
```

**num**

A unique number for each of the 21 different built-in types.

These are roughly ordered from least-to-most precision.

**shape**

Shape tuple of the sub-array if this data type describes a sub-array, and ``()`` otherwise.

**str**

The array-protocol typestring of this data-type object.

**subtype**

Tuple ``(item\_dtype, shape)`` if this `dtype` describes a sub-array, and None otherwise.

The \*shape\* is the fixed shape of the sub-array described by this data type, and \*item\_dtype\* the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by \*shape\* are tacked on to the end of the retrieved array.

**type**

The type object used to instantiate a scalar of this data-type.

Data and other attributes defined here:

**\_new\_** = <built-in method `_new_` of type object>  
`T. _new_(S, ...)` -> a new `object` with type S, a subtype of T

```
class errstate(__builtin__.object)
 errstate(**kwargs)
```

Context manager for `floating`-point error handling.

Using an instance of `errstate` as a context manager allows statements in that context to execute with a known error handling behavior. Upon entering the context the error handling is set with `seterr` and `seterrcall`, and upon exiting it is reset to what it was before.

**Parameters**  
-----  
**kwargs** : {divide, over, under, invalid}  
 Keyword arguments. The valid keywords are the possible [floating-point](#) exceptions. Each keyword should have a string value that defines the treatment for the particular error. Possible values are {'ignore', 'warn', 'raise', 'call', 'print', 'log'}.

**See Also**  
-----  
[seterr](#), [geterr](#), [seterrcall](#), [geterrcall](#)

**Notes**  
-----  
 The ``with`` statement was introduced in Python 2.5, and can only be used there by importing it: ``from \_\_future\_\_ import with\_statement``. In earlier Python versions the ``with`` statement is not available.

For complete documentation of the types of [floating-point](#) exceptions and treatment options, see `seterr`.

**Examples**  
-----  
`>>> from __future__ import with_statement # use 'with' in Python 2.5  
>>> olderr = np.seterr(all='ignore') # Set error handling to known state.`  
`>>> np.arange(3) / 0.  
array([NaN, Inf, Inf])  
>>> with np.errstate(divide='warn'):  
... np.arange(3) / 0.  
...  
__main__:2: RuntimeWarning: divide by zero encountered in divide  
array([NaN, Inf, Inf])`  
`>>> np.sqrt(-1)  
nan  
>>> with np.errstate(invalid='raise'):  
... np.sqrt(-1)  
Traceback (most recent call last):  
 File "<stdin>", line 2, in <module>  
FloatingPointError: invalid value encountered in sqrt`  
 Outside the context the error handling behavior has not changed:  
`>>> np.geterr()  
{'over': 'warn', 'divide': 'warn', 'invalid': 'warn',  
'under': 'ignore'}`

Methods defined here:

[\\_\\_enter\\_\\_](#)(self)  
[\\_\\_exit\\_\\_](#)(self, \*exc\_info)  
[\\_\\_init\\_\\_](#)(self, \*\*kwargs)

---

Data descriptors defined here:

[\\_\\_dict\\_\\_](#)  
 dictionary for instance variables (if defined)  
[\\_\\_weakref\\_\\_](#)  
 list of weak references to the object (if defined)

---

class [finfo](#)([\\_builtin\\_.object](#))  
[finfo](#)([dtype](#))

Machine limits for [floating](#) point types.

**Attributes**  
-----  
**eps** : [float](#)  
 The smallest representable positive [number](#) such that  
 ``1.0 + eps != 1.0``. Type of `eps` is an appropriate [floating](#) point type.  
**epsneg** : [floating](#) point [number](#) of the appropriate type  
 The smallest representable positive [number](#) such that  
 ``1.0 - epsneg != 1.0``.  
**iexp** : [int](#)  
 The [number](#) of bits in the exponent portion of the [floating](#) point representation.

```

machar : MachAr
 The object which calculated these parameters and holds more
 detailed information.
machep : int
 The exponent that yields `eps`.
max : floating point number of the appropriate type
 The largest representable number.
maxexp : int
 The smallest positive power of the base (2) that causes overflow.
min : floating point number of the appropriate type
 The smallest representable number, typically ``-max``.
minexp : int
 The most negative power of the base (2) consistent with there
 being no leading 0's in the mantissa.
negep : int
 The exponent that yields `epsneg`.
nexp : int
 The number of bits in the exponent including its sign and bias.
nmant : int
 The number of bits in the mantissa.
precision : int
 The approximate number of decimal digits to which this kind of
 float is precise.
resolution : floating point number of the appropriate type
 The approximate decimal resolution of this type, i.e.,
 ``10**-precision``.
tiny : float
 The smallest positive usable number. Type of `tiny` is an
 appropriate floating point type.

```

**Parameters**

[dtype](#) : [float](#), [dtype](#), or instance  
     Kind of [floating](#) point data-type about which to get information.

**See Also**

[MachAr](#) : The implementation of the tests that produce this information.  
[iinfo](#) : The equivalent for [integer](#) data types.

**Notes**

For developers of NumPy: do not instantiate this at the module level.  
The initial calculation of these parameters is expensive and negatively  
impacts import times. These objects are cached, so calling ``[finfo\(\)](#)``  
repeatedly inside your functions is not a problem.

Methods defined here:

[\\_\\_repr\\_\\_](#)(self)  
[\\_\\_str\\_\\_](#)(self)

---

Static methods defined here:

[\\_\\_new\\_\\_](#)(cls, dtype)

---

Data descriptors defined here:

[\\_\\_dict\\_\\_](#)  
     dictionary for instance variables (if defined)  
[\\_\\_weakref\\_\\_](#)  
     list of weak references to the object (if defined)

---

class [flatiter](#)([\\_builtin\\_.object](#))
 Flat iterator [object](#) to iterate over arrays.

 A `[flatiter](#)` iterator is returned by ``x.flat`` for any array `x`.
 It allows iterating over the array as if it were a 1-D array,
 either in a for-loop or by calling its `next` method.

 Iteration is done in row-major, C-style order (the last
 index varying the fastest). The iterator can also be indexed using
 basic slicing or advanced indexing.

**See Also**

[ndarray.flat](#) : Return a flat iterator over an array.  
[ndarray.flatten](#) : Returns a flattened copy of an array.

**Notes**

-----  
A `flatiter` iterator can not be constructed directly from Python code  
by calling the `flatiter` constructor.

**Examples**

-----  
`>>> x = np.arange(6).reshape(2, 3)  
>>> fl = x.flat  
>>> type(fl)  
<type 'numpy.flatiter'>  
>>> for item in fl:  
... print(item)  
...  
0  
1  
2  
3  
4  
5  
  
>>> fl[2:4]  
array([2, 3])`

Methods defined here:

```

__array__(...)
 array(type=None) Get array from iterator

__delitem__(...)
 x. delitem(y) <==> del x[y]

__eq__(...)
 x. eq(y) <==> x==y

__ge__(...)
 x. ge(y) <==> x>=y

__getitem__(...)
 x. getitem(y) <==> x[y]

__gt__(...)
 x. gt(y) <==> x>y

__iter__(...)
 x. iter() <==> iter(x)

__le__(...)
 x. le(y) <==> x<=y

__len__(...)
 x. len() <==> len(x)

__lt__(...)
 x. lt(y) <==> x<y

__ne__(...)
 x. ne(y) <==> x!=y

__setitem__(...)
 x. setitem(i, y) <==> x[i]=y

copy(...)
 copy()

 Get a copy of the iterator as a 1-D array.

 Examples

 >>> x = np.arange(6).reshape(2, 3)
 >>> x
 array([[0, 1, 2],
 [3, 4, 5]])
 >>> fl = x.flat
 >>> fl.copy()
 array([0, 1, 2, 3, 4, 5])

next(...)
 x.next() -> the next value, or raise StopIteration

```

Data descriptors defined here:

### **base**

A reference to the array that is iterated over.

Examples

```

 >>> x = np.arange(5)
 >>> fl = x.flat
 >>> fl.base is x
 True

```

### **coords**

An N-dimensional tuple of current coordinates.

Examples

```

 >>> x = np.arange(6).reshape(2, 3)
 >>> fl = x.flat
 >>> fl.coords
 (0, 0)
 >>> fl.next()

```

```
 0
>>> fl.coords
(0, 1)

index
 Current flat index into the array.

Examples

>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> fl.index
0
>>> fl.next()
0
>>> fl.index
1
```

---

```
class flexible(generic)
Method resolution order:
 flexible
 generic
 builtin_object
```

---

Methods inherited from generic:

```
abs(...)
x. abs() <==> abs(x)

add(...)
x. add(y) <==> x+y

and(...)
x. and(y) <==> x&y

array(...)
sc. array(|type) return 0-dim array

_array_wrap_(...)
sc. array_wrap(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
x. div(y) <==> x/y

divmod(...)
x. divmod(y) <==> divmod(x, y)

eq(...)
x. eq(y) <==> x==y

float(...)
x. float() <==> float(x)

floordiv(...)
x. floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

ge(...)
x. ge(y) <==> x>=y

getitem(...)
x. getitem(y) <==> x[y]

gt(...)
x. gt(y) <==> x>y

hex(...)
```

```
x.hex() <==> hex(x)

int(...)
x.int() <==> int(x)

invert(...)
x.invert() <==> ~x

le(...)
x.le(y) <==> x<=y

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

lt(...)
x.lt(y) <==> x<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

ne(...)
x.ne(y) <==> x!=y

neg(...)
x.neg() <==> -x

nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
```

```
x.rmul(y) <==> y*x
ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y

truediv(...)
x.truediv(y) <==> x/y

xor(...)
x.xor(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

```
new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

>Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**

```

 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

class float128(floating)
128-bit floating-point number. Character code: 'g'. C long float
compatible.
```

Method resolution order:

```

float128
floating
inexact
number
generic
builtin .object
```

---

Methods defined here:

```

__eq__(...)
 x. __eq__(y) <==> x==y

__float__(...)
 x. __float__() <==> float(x)

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__hash__(...)
 x. __hash__() <==> hash(x)

__hex__(...)
 x. __hex__() <==> hex(x)

__int__(...)
 x. __int__() <==> int(x)

__le__(...)
 x. __le__(y) <==> x<=y

__long__(...)
 x. __long__() <==> long(x)

__lt__(...)
 x. __lt__(y) <==> x<y

__ne__(...)
 x. __ne__(y) <==> x!=y

__oct__(...)
 x. __oct__() <==> oct(x)

__repr__(...)
 x. __repr__() <==> repr(x)

__str__(...)
 x. __str__() <==> str(x)
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
```

T.new(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from generic:

abs(...)  
x.abs() <==> abs(x)

add(...)  
x.add(y) <==> x+y

and(...)  
x.and(y) <==> x&y

array(...)  
sc.array(|type) return 0-dim array

array\_wrap(...)  
sc.array\_wrap(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)  
x.div(y) <==> x/y

divmod(...)  
x.divmod(y) <==> divmod(x, y)

floordiv(...)  
x.floordiv(y) <==> x//y

format(...)  
NumPy scalar formatter

getitem(...)  
x.getitem(y) <==> x[y]

invert(...)  
x.invert() <==> ~x

lshift(...)  
x.lshift(y) <==> x<<y

mod(...)  
x.mod(y) <==> x%y

mul(...)  
x.mul(y) <==> x\*y

neg(...)  
x.neg() <==> -x

nonzero(...)  
x.nonzero() <==> x != 0

or(...)  
x.or(y) <==> x|y

pos(...)  
x.pos() <==> +x

pow(...)  
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)  
x.radd(y) <==> y+x

rand(...)  
x.rand(y) <==> y&x

rdiv(...)  
x.rdiv(y) <==> y/x

```
__rdivmod__(...)
 x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
 x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
 x.__rlshift__(y) <==> y<<x

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(y) <==> y*x

__ror__(...)
 x.__ror__(y) <==> y|x

__rpow__(...)
 y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x.__rrshift__(y) <==> y>>x

__rshift__(...)
 x.__rshift__(y) <==> x>>y

__rsub__(...)
 x.__rsub__(y) <==> y-x

__rtruediv__(...)
 x.__rtruediv__(y) <==> y/x

__rxor__(...)
 x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
 x.__sub__(y) <==> x-y

__truediv__(...)
 x.__truediv__(y) <==> x/y

__xor__(...)
 x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
```

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

```
newbyteorder(new_order='S')

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

repeat(...)

 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

**class float16([floating](#))**

Method resolution order:

[float16](#)  
[floating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[builtin](#) [object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_](#)(...)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_ge\\_\\_](#)(...)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_](#)(...)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_](#)(...)  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_le\\_\\_](#)(...)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_lt\\_\\_](#)(...)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_](#)(...)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

[\\_\\_repr\\_\\_](#)(...)  
x. [\\_\\_repr\\_\\_](#)() <==> repr(x)

[\\_\\_str\\_\\_](#)(...)  
x. [\\_\\_str\\_\\_](#)() <==> [str](#)(x)

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [generic](#):

```
__abs__(...)
x.__abs__() <==> abs(x)

__add__(...)
x.__add__(y) <==> x+y

__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
```

```
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

sub(...)
x.sub(y) <==> x-y

truediv(...)
x.truediv(y) <==> x/y

xor(...)
x.xor(y) <==> x^y

all(...)
Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str`, optional

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**put(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

```
dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

```
class float32(floating)
 32-bit floating-point number. Character code 'f'. C float compatible.
```

Method resolution order:

```
float32
floating
inexact
number
generic
_builtin__object
```

---

Methods defined here:

```
__eq__(...)
 x. __eq__(y) <==> x==y

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__hash__(...)
 x. __hash__() <==> hash(x)

__le__(...)
 x. __le__(y) <==> x<=y

__lt__(...)
 x. __lt__(y) <==> x<y

__ne__(...)
 x. __ne__(y) <==> x!=y

__repr__(...)
 x. __repr__() <==> repr(x)
```

**\_\_str\_\_**(...)  
x.\_\_str\_\_() <==> str(x)

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from generic:

**\_\_abs\_\_**(...)  
x.\_\_abs\_\_() <==> abs(x)  
  
**\_\_add\_\_**(...)  
x.\_\_add\_\_(y) <==> x+y  
  
**\_\_and\_\_**(...)  
x.\_\_and\_\_(y) <==> x&y  
  
**\_\_array\_\_**(...)  
sc.\_\_array\_\_(|type) return 0-dim array  
  
**\_\_array\_wrap\_\_**(...)  
sc.\_\_array\_wrap\_\_(obj) return scalar from array  
  
**\_\_copy\_\_**(...)  
  
**\_\_deepcopy\_\_**(...)  
  
**\_\_div\_\_**(...)  
x.\_\_div\_\_(y) <==> x/y  
  
**\_\_divmod\_\_**(...)  
x.\_\_divmod\_\_(y) <==> divmod(x, y)  
  
**\_\_float\_\_**(...)  
x.\_\_float\_\_() <==> float(x)  
  
**\_\_floordiv\_\_**(...)  
x.\_\_floordiv\_\_(y) <==> x//y  
  
**\_\_format\_\_**(...)  
NumPy array scalar formatter  
  
**\_\_getitem\_\_**(...)  
x.\_\_getitem\_\_(y) <==> x[y]  
  
**\_\_hex\_\_**(...)  
x.\_\_hex\_\_() <==> hex(x)  
  
**\_\_int\_\_**(...)  
x.\_\_int\_\_() <==> int(x)  
  
**\_\_invert\_\_**(...)  
x.\_\_invert\_\_() <==> ~x  
  
**\_\_long\_\_**(...)  
x.\_\_long\_\_() <==> long(x)  
  
**\_\_lshift\_\_**(...)  
x.\_\_lshift\_\_(y) <==> x<<y  
  
**\_\_mod\_\_**(...)  
x.\_\_mod\_\_(y) <==> x%y  
  
**\_\_mul\_\_**(...)  
x.\_\_mul\_\_(y) <==> x\*y  
  
**\_\_neg\_\_**(...)  
x.\_\_neg\_\_() <==> -x  
  
**\_\_nonzero\_\_**(...)  
x.\_\_nonzero\_\_() <==> x != 0

```
oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

sub(...)
x.sub(y) <==> x-y

truediv(...)
x.truediv(y) <==> x/y

xor(...)
x.xor(y) <==> x^y

all(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**any(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmax(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

```

__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

```

---

```

class float64(floating, __builtin__.float)
64-bit floating-point number. Character code 'd'. Python float compatible.

```

Method resolution order:

```

float64
floating
inexact
number
generic
__builtin__.float
__builtin__.object

```

---

Methods defined here:

```

__eq__(...)
x. __eq__(y) <==> x==y

__ge__(...)
x. __ge__(y) <==> x>=y

__gt__(...)
x. __gt__(y) <==> x>y

__le__(...)
x. __le__(y) <==> x<=y

__lt__(...)
x. __lt__(y) <==> x<y

```

---

```
__ne__(...)
x.__ne__(y) <==> x!=y

__repr__(...)
x.__repr__() <==> repr(x)

__str__(...)
x.__str__() <==> str(x)
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

---

Methods inherited from [generic](#):

```
__abs__(...)
x.__abs__() <==> abs(x)

__add__(...)
x.__add__(y) <==> x+y

__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type|) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y
```

```
__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y
```

**\_\_xor\_\_(...)**  
x.xor(y) <=> x^y

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **conj(...)**

#### **conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.
```

```
See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

resize(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

round(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

searchsorted(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

setfield(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)****tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

```

__array_interface__
 Array protocol: Python side

__array_priority__
 Array priority.

__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

```

---

Methods inherited from [\\_\\_builtin\\_\\_.float](#):

```

__coerce__(...)
 x. __coerce__(y) <==> coerce(x, y)

__getattribute__(...)
 x. __getattribute__('name') <==> x.name

__getformat__(...)
 float.__getformat__(typestr) -> string

 You probably don't want to use this function. It exists mainly to be
 used in Python's test suite.

 typestr must be 'double' or 'float'. This function returns whichever of
 'unknown', 'IEEE, big-endian' or 'IEEE, little-endian' best describes the
 format of floating point numbers used by the C type named by typestr.

__getnewargs__(...)

__hash__(...)
 x. __hash__() <==> hash(x)

__setformat__(...)
 float.__setformat__(typestr, fmt) -> None

 You probably don't want to use this function. It exists mainly to be

```

```

 used in Python's test suite.

 typestr must be 'double' or 'float'. fmt must be one of 'unknown',
 'IEEE, big-endian' or 'IEEE, little-endian', and in addition can only be
 one of the latter two if it appears to match the underlying C reality.

 Override the automatic determination of C-level floating point type.
 This affects how floats are converted to and from binary strings.

trunc(...)
 Return the Integral closest to x between 0 and x.

as_integer_ratio(...)
float.as_integer_ratio\(\) -> (int, int)
 Return a pair of integers, whose ratio is exactly equal to the original
 float and with a positive denominator.
 Raise OverflowError on infinities and a ValueError on NaNs.

 >>> (10.0).as_integer_ratio()
 (10, 1)
 >>> (0.0).as_integer_ratio()
 (0, 1)
 >>> (-.25).as_integer_ratio()
 (-1, 4)

fromhex(...)
float.fromhex\(string\) -> float
 Create a floating-point number from a hexadecimal string.
 >>> float.fromhex('0x1.ffffp10')
 2047.984375
 >>> float.fromhex('-0x1p-1074')
 -4.9406564584124654e-324

hex(...)
float.hex\(\) -> string
 Return a hexadecimal representation of a floating-point number.
 >>> (-0.1).hex()
 '-0x1.999999999999ap-4'
 >>> 3.14159.hex()
 '0x1.921f9f01b866ep+1'

is_integer(...)
 Return True if the float is an integer.
```

---

**float** = class float64([floating](#), [builtin .float](#))  
 64-bit [floating-point number](#). Character code 'd'. Python [float](#) compatible.

Method resolution order:

```

float64
floating
inexact
number
generic
builtin .float
builtin .object

```

---

Methods defined here:

```

eq(...)
 x. _eq_(y) <==> x==y

ge(...)
 x. _ge_(y) <==> x>=y

gt(...)
 x. _gt_(y) <==> x>y

le(...)
 x. _le_(y) <==> x<=y

lt(...)
 x. _lt_(y) <==> x<y

```

```
__ne__(...)
x.__ne__(y) <==> x!=y

__repr__(...)
x.__repr__() <==> repr(x)

__str__(...)
x.__str__() <==> str(x)
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

---

Methods inherited from [generic](#):

```
__abs__(...)
x.__abs__() <==> abs(x)

__add__(...)
x.__add__(y) <==> x+y

__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type|) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y
```

```
__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y
```

**\_\_xor\_\_(...)**  
x.xor(y) <=> x^y

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **conj(...)**

### **conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)****tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

```
__array_interface__
 Array protocol: Python side

__array_priority__
 Array priority.

__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

---

Methods inherited from [\\_\\_builtin\\_\\_.float](#):

```
__coerce__(...)
 x. __coerce__(y) <==> coerce(x, y)

__getattribute__(...)
 x. __getattribute__('name') <==> x.name

__getformat__(...)
 float.__getformat__(typestr) -> string

 You probably don't want to use this function. It exists mainly to be
 used in Python's test suite.

 typestr must be 'double' or 'float'. This function returns whichever of
 'unknown', 'IEEE, big-endian' or 'IEEE, little-endian' best describes the
 format of floating point numbers used by the C type named by typestr.

__getnewargs__(...)

__hash__(...)
 x. __hash__() <==> hash(x)

__setformat__(...)
 float.__setformat__(typestr, fmt) -> None

 You probably don't want to use this function. It exists mainly to be
```

```

 used in Python's test suite.

 typestr must be 'double' or 'float'. fmt must be one of 'unknown',
 'IEEE, big-endian' or 'IEEE, little-endian', and in addition can only be
 one of the latter two if it appears to match the underlying C reality.

 Override the automatic determination of C-level floating point type.
 This affects how floats are converted to and from binary strings.

trunc(...)
 Return the Integral closest to x between 0 and x.

as_integer_ratio(...)
float.as_integer_ratio\(\) -> (int, int)
 Return a pair of integers, whose ratio is exactly equal to the original
 float and with a positive denominator.
 Raise OverflowError on infinities and a ValueError on NaNs.

 >>> (10.0).as_integer_ratio()
 (10, 1)
 >>> (0.0).as_integer_ratio()
 (0, 1)
 >>> (-.25).as_integer_ratio()
 (-1, 4)

fromhex(...)
float.fromhex\(string\) -> float
 Create a floating-point number from a hexadecimal string.
 >>> float.fromhex\('0x1.ffffp10'\)
 2047.984375
 >>> float.fromhex\('-0x1p-1074'\)
 -4.9406564584124654e-324

hex(...)
float.hex\(\) -> string
 Return a hexadecimal representation of a floating-point number.
 >>> (-0.1).hex()
 '-0x1.999999999999ap-4'
 >>> 3.14159.hex()
 '0x1.921f9f01b866ep+1'

is_integer(...)
 Return True if the float is an integer.
```

---

**class [floating](#)([inexact](#))**

Method resolution order:

```

floating
inexact
number
generic
builtin.object

```

---

Methods inherited from [generic](#):

```

abs(...)
 x._abs_() <==> abs(x)

add(...)
 x._add_(y) <==> x+y

and(...)
 x._and_(y) <==> x&y

array(...)
 sc._array_(|type) return 0-dim array

_array_wrap_(...)
 sc._array_wrap_(obj) return scalar from array

copy(...)

deepcopy(...)

```

```
div(...)
x._div_(y) <==> x/y

divmod(...)
x._divmod_(y) <==> divmod(x, y)

eq(...)
x._eq_(y) <==> x==y

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

ge(...)
x._ge_(y) <==> x>=y

getitem(...)
x._getitem_(y) <==> x[y]

gt(...)
x._gt_(y) <==> x>y

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

le(...)
x._le_(y) <==> x<=y

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

lt(...)
x._lt_(y) <==> x<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

ne(...)
x._ne_(y) <==> x!=y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])
```

```
__radd__(...)
x. __radd__(y) <==> y+x

__rand__(...)
x. __rand__(y) <==> y&x

__rdiv__(...)
x. __rdiv__(y) <==> y/x

__rdivmod__(...)
x. __rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x. __repr__() <==> repr(x)

__rfloordiv__(...)
x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
x. __rlshift__(y) <==> y<<x

__rmod__(...)
x. __rmod__(y) <==> y%x

__rmul__(...)
x. __rmul__(y) <==> y*x

__ror__(...)
x. __ror__(y) <==> y|x

__rpow__(...)
y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x. __rrshift__(y) <==> y>>x

__rshift__(...)
x. __rshift__(y) <==> x>>y

__rsub__(...)
x. __rsub__(y) <==> y-x

__rtruediv__(...)
x. __rtruediv__(y) <==> y/x

__rxor__(...)
x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x. __str__() <==> str(x)

__sub__(...)
x. __sub__(y) <==> x-y

__truediv__(...)
x. __truediv__(y) <==> x/y

__xor__(...)
x. __xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**any(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmax(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**repeat(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**reshape(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to

provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**

```
base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

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

 Attributes

 dtype : dtype
 The converted data-type.

 Parameters

 formats : str or list of str
 The format description, either specified as a string with
 comma-separated format descriptions in the form ``'f8, i4, a5'``, or
 a list of format description strings in the form
 ``['f8', 'i4', 'a5']``.
 names : str or list/tuple of str
 The field names, either specified as a comma-separated string in the
 form ``'col1, col2, col3'``, or as a list or tuple of strings in the
 form ``['col1', 'col2', 'col3']``.
 An empty list can be used, in that case default field names
 ('f0', 'f1', ...) are used.
 titles : sequence
 Sequence of title strings. An empty list can be used to leave titles
 out.
 aligned : bool, optional
 If True, align the fields by padding as the C-compiler would.
 Default is False.
 byteorder : str, optional
 If specified, all the fields will be changed to the
 provided byte-order. Otherwise, the default byte-order is
 used. For all available string specifiers, see ``dtype.newbyteorder``.

 See Also

 dtype, typename, sctype2char
```

```

Examples

>>> np.format_parser(['f8', 'i4', 'a5'], ['col1', 'col2', 'col3'],
... ['T1', 'T2', 'T3']).dtype
dtype([('T1', 'col1'), ('<f8'), ('T2', 'col2'), ('<i4'),
 ('T3', 'col3'), ('|S5'))]

`names` and/or `titles` can be empty lists. If `titles` is an empty list,
titles will simply not appear. If `names` is empty, default field names
will be used.

>>> np.format_parser(['f8', 'i4', 'a5'], ['col1', 'col2', 'col3'],
... []).dtype
dtype([('col1', '<f8'), ('col2', '<i4'), ('col3', '|S5'))]
>>> np.format_parser(['f8', 'i4', 'a5'], [], []).dtype
dtype([('f0', '<f8'), ('f1', '<i4'), ('f2', '|S5'))]

```

Methods defined here:

`__init__(self, formats, names, titles, aligned=False, byteorder=None)`

```

class generic(__builtin__.object)
 Base class for numpy scalar types.

 Class from which most (all?) numpy scalar types are derived. For
 consistency, exposes the same API as `ndarray`, despite many
 consequent attributes being either "get-only," or completely irrelevant.
 This is the class from which it is strongly suggested users should derive
 custom scalar types.

```

Methods defined here:

```

__abs__(...)
 x.__abs__() <==> abs(x)

__add__(...)
 x.__add__(y) <==> x+y

__and__(...)
 x.__and__(y) <==> x&y

__array__(...)
 sc.__array__(|type|) return 0-dim array

__array_wrap__(...)
 sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x.__div__(y) <==> x/y

__divmod__(...)
 x.__divmod__(y) <==> divmod(x, y)

__eq__(...)
 x.__eq__(y) <==> x==y

__float__(...)
 x.__float__() <==> float(x)

__floordiv__(...)
 x.__floordiv__(y) <==> x//y

__format__(...)
 NumPy array scalar formatter

__ge__(...)
 x.__ge__(y) <==> x>=y

__getitem__(...)
 x.__getitem__(y) <==> x[y]

```

```
__gt__(...)
x.__gt__(y) <==> x>y

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__le__(...)
x.__le__(y) <==> x<=y

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__lt__(...)
x.__lt__(y) <==> x<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__ne__(...)
x.__ne__(y) <==> x!=y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x
```

```
__rmod__(...)
 x. __rmod__(y) <==> y%x

__rmul__(...)
 x. __rmul__(y) <==> y*x

__ror__(...)
 x. __ror__(y) <==> y|x

__rpow__(...)
 y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x. __rrshift__(y) <==> y>>x

__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian

```
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors defined here:

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

**half = class float16([floating](#))**

Method resolution order:

```
float16
floating
inexact
number
generic
 builtin .object
```

---

Methods defined here:

**\_\_eq\_\_(...)**  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

**\_\_ge\_\_(...)**  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

**\_\_gt\_\_(...)**  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

**\_\_hash\_\_(...)**  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

**\_\_le\_\_(...)**  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

**\_\_lt\_\_(...)**  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

**\_\_ne\_\_(...)**  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

**\_\_repr\_\_(...)**  
x. [\\_\\_repr\\_\\_](#)() <==> repr(x)

**\_\_str\_\_(...)**  
x. [\\_\\_str\\_\\_](#)() <==> [str](#)(x)

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x. [\\_\\_abs\\_\\_](#)() <==> abs(x)

**\_\_add\_\_(...)**  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y

**\_\_and\_\_(...)**  
x. [\\_\\_and\\_\\_](#)(y) <==> x&y

```
__array__(...)
 sc.__array__(|type) return 0-dim array

__array_wrap__(...)
 sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x.__div__(y) <==> x/y

__divmod__(...)
 x.__divmod__(y) <==> divmod(x, y)

__float__(...)
 x.__float__() <==> float(x)

__floordiv__(...)
 x.__floordiv__(y) <==> x//y

__format__(...)
 NumPy array scalar formatter

__getitem__(...)
 x.__getitem__(y) <==> x[y]

__hex__(...)
 x.__hex__() <==> hex(x)

__int__(...)
 x.__int__() <==> int(x)

__invert__(...)
 x.__invert__() <==> ~x

__long__(...)
 x.__long__() <==> long(x)

__lshift__(...)
 x.__lshift__(y) <==> x<<y

__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(y) <==> x*y

__neg__(...)
 x.__neg__() <==> -x

__nonzero__(...)
 x.__nonzero__() <==> x != 0

__oct__(...)
 x.__oct__() <==> oct(x)

__or__(...)
 x.__or__(y) <==> x|y

__pos__(...)
 x.__pos__() <==> +x

__pow__(...)
 x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
 x.__radd__(y) <==> y+x

__rand__(...)
 x.__rand__(y) <==> y&x

__rdiv__(...)
```

```
x.__rdiv__(y) <==> y/x
__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)
__reduce__(...)
__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x
__rlshift__(...)
x.__rlshift__(y) <==> y<<x
__rmod__(...)
x.__rmod__(y) <==> y%x
__rmul__(...)
x.__rmul__(y) <==> y*x
__ror__(...)
x.__ror__(y) <==> y|x
__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])
__rrshift__(...)
x.__rrshift__(y) <==> y>>x
__rshift__(...)
x.__rshift__(y) <==> x>>y
__rsub__(...)
x.__rsub__(y) <==> y-x
__rtruediv__(...)
x.__rtruediv__(y) <==> y/x
__rxor__(...)
x.__rxor__(y) <==> y^x
__setstate__(...)
__sizeof__(...)
__sub__(...)
x.__sub__(y) <==> x-y
__truediv__(...)
x.__truediv__(y) <==> x/y
__xor__(...)
x.__xor__(y) <==> x^y
all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

```
newbyteorder(new_order='S')

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

repeat(...)

 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

```
class iinfo(_builtin_.object)
 iinfo(type)

 Machine limits for integer types.

 Attributes

 min : int
 The smallest integer expressible by the type.
 max : int
 The largest integer expressible by the type.

 Parameters

 int_type : integer type, dtype, or instance
 The kind of integer data type to get information about.

 See Also

 finfo : The equivalent for floating point data types.

 Examples

 With types:

 >>> ii16 = np.iinfo(np.int16)
 >>> ii16.min
 -32768
 >>> ii16.max
 32767
 >>> ii32 = np.iinfo(np.int32)
 >>> ii32.min
 -2147483648
 >>> ii32.max
 2147483647

 With instances:

 >>> ii32 = np.iinfo(np.int32(10))
 >>> ii32.min
 -2147483648
 >>> ii32.max
 2147483647

 Methods defined here:

 __init__(self, int_type)
 __repr__(self)
 __str__(self)
 String representation.
```

---

Data descriptors defined here:

```
dict
 dictionary for instance variables (if defined)

weakref
 list of weak references to the object (if defined)

max
 Maximum value of given dtype.

min
 Minimum value of given dtype.
```

```
class inexact(number)
 Method resolution order:
 inexact
 number
 generic
 builtin object
```

---

Methods inherited from [generic](#):

```
abs(...)
 x._abs_() <==> abs(x)

add(...)
 x._add_(y) <==> x+y

and(...)
 x._and_(y) <==> x&y

array(...)
 sc._array_(|type|) return 0-dim array

_array_wrap_(...)
 sc._array_wrap_(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
 x._div_(y) <==> x/y

divmod(...)
 x._divmod_(y) <==> divmod(x, y)

eq(...)
 x._eq_(y) <==> x==y

float(...)
 x._float_() <==> float(x)

floordiv(...)
 x._floordiv_(y) <==> x//y

format(...)
 NumPy array scalar formatter

ge(...)
 x._ge_(y) <==> x>=y

getitem(...)
 x._getitem_(y) <==> x[y]

gt(...)
 x._gt_(y) <==> x>y

hex(...)
 x._hex_() <==> hex(x)
```

```
__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__le__(...)
x.__le__(y) <==> x<=y

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__lt__(...)
x.__lt__(y) <==> x<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__ne__(...)
x.__ne__(y) <==> x!=y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x
```

```
__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----  
new\_order : [str](#), optional  
Byte order to force; a value from the [byte](#) order specifications

above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----  
`new_dtype : dtype`  
New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
resize(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
round(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
searchsorted(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
setfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
setflags(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
sort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
squeeze(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
std(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
```

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **tobytes(...)**

#### **tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

#### **tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

#### **toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

#### **trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

**\_\_array\_interface\_\_**

Array protocol: Python side

**\_\_array\_priority\_\_**

Array priority.

**\_\_array\_struct\_\_**

Array protocol: struct

**base**

base object

**data**

pointer to start of data

**dtype**

get array data-descriptor

**flags**

integer value of flags

**flat**

a 1-d view of scalar

**imag**

imaginary part of scalar

**itemsize**

length of one element in bytes

**nbytes**

length of item in bytes

**ndim**

number of array dimensions

**real**

```

real part of scalar

shape
tuple of array dimensions

size
number of elements in the gentype

strides
tuple of bytes steps in each dimension

```

---

```

int0 = class int64(signedinteger, builtin_int)
64-bit integer. Character code 'l'. Python int compatible.

```

Method resolution order:

```

int64
signedinteger
integer
number
generic
builtin_int
builtin_object

```

---

Methods defined here:

```

__eq__\(...\)
x. __eq__\(y\) <==> x==y

__ge__\(...\)
x. __ge__\(y\) <==> x>=y

__gt__\(...\)
x. __gt__\(y\) <==> x>y

__index__\(...\)
x[y:z] <==> x[y. __index__\(\):z. __index__\(\)]

__le__\(...\)
x. __le__\(y\) <==> x<=y

__lt__\(...\)
x. __lt__\(y\) <==> x<y

__ne__\(...\)
x. __ne__\(y\) <==> x!=y

```

---

Data and other attributes defined here:

```

__new__ = <built-in method __new__ of type object>
T. __new__\(S, ...\) -> a new object with type S, a subtype of T

```

---

Data descriptors inherited from [integer](#):

```

denominator
denominator of value (1)

numerator
numerator of value (the value itself)

```

---

Methods inherited from [generic](#):

```

__abs__\(...\)
x. __abs__\(\) <==> abs(x)

__add__\(...\)
x. __add__\(y\) <==> x+y

__and__\(...\)
x. __and__\(y\) <==> x&y

```

```
__array__(...)
 sc.__array__(|type) return 0-dim array

__array_wrap__(...)
 sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x.__div__(y) <==> x/y

__divmod__(...)
 x.__divmod__(y) <==> divmod(x, y)

__float__(...)
 x.__float__() <==> float(x)

__floordiv__(...)
 x.__floordiv__(y) <==> x//y

__format__(...)
 NumPy array scalar formatter

__getitem__(...)
 x.__getitem__(y) <==> x[y]

__hex__(...)
 x.__hex__() <==> hex(x)

__int__(...)
 x.__int__() <==> int(x)

__invert__(...)
 x.__invert__() <==> ~x

__long__(...)
 x.__long__() <==> long(x)

__lshift__(...)
 x.__lshift__(y) <==> x<<y

__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(y) <==> x*y

__neg__(...)
 x.__neg__() <==> -x

__nonzero__(...)
 x.__nonzero__() <==> x != 0

__oct__(...)
 x.__oct__() <==> oct(x)

__or__(...)
 x.__or__(y) <==> x|y

__pos__(...)
 x.__pos__() <==> +x

__pow__(...)
 x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
 x.__radd__(y) <==> y+x

__rand__(...)
 x.__rand__(y) <==> y&x

__rdiv__(...)
```

```
x.__rdiv__(y) <==> y/x
__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%
__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

`newbyteorder(new_order='S')`

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**put(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**repeat(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**reshape(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**

```

 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

```

---

Methods inherited from [builtin\\_int](#):

```

__cmp__(...)
 x.cmp(y) <==> cmp(x,y)

__coerce__(...)
 x.coerce(y) <==> coerce(x, y)

__getattribute__(...)
 x.getattribute('name') <==> x.name

__getnewargs__(...)

__hash__(...)
 x.hash() <==> hash(x)

__trunc__(...)
 Truncating an Integral returns itself.

bit_length(...)
int.bit_length\(\) -> int

 Number of bits necessary to represent self in binary.
 >>> bin(37)
 '0b100101'
 >>> (37).bit_length()
 6

```

---

```

class int16(signedinteger)
 16-bit integer. Character code ``h''. C short compatible.

```

Method resolution order:

```

int16
signedinteger
integer
number
generic
builtin_object

```

---

Methods defined here:

---

**\_\_eq\_\_**(...)  
x. \_\_eq\_\_(y) <==> x==y

**\_\_ge\_\_**(...)  
x. \_\_ge\_\_(y) <==> x>=y

**\_\_gt\_\_**(...)  
x. \_\_gt\_\_(y) <==> x>y

**\_\_hash\_\_**(...)  
x. \_\_hash\_\_() <==> hash(x)

**\_\_index\_\_**(...)  
x[y:z] <==> x[y. \_\_index\_\_():z. \_\_index\_\_()]

**\_\_le\_\_**(...)  
x. \_\_le\_\_(y) <==> x<=y

**\_\_lt\_\_**(...)  
x. \_\_lt\_\_(y) <==> x<y

**\_\_ne\_\_**(...)  
x. \_\_ne\_\_(y) <==> x!=y

---

Data and other attributes defined here:

---

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T. \_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

---

Data descriptors inherited from integer:

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from generic:

---

**\_\_abs\_\_**(...)  
x. \_\_abs\_\_() <==> abs(x)

**\_\_add\_\_**(...)  
x. \_\_add\_\_(y) <==> x+y

**\_\_and\_\_**(...)  
x. \_\_and\_\_(y) <==> x&y

**\_\_array\_\_**(...)  
sc. \_\_array\_\_(|type) return 0-dim array

**\_\_array\_wrap\_\_**(...)  
sc. \_\_array\_wrap\_\_(obj) return scalar from array

**\_\_copy\_\_**(...)

**\_\_deepcopy\_\_**(...)

**\_\_div\_\_**(...)  
x. \_\_div\_\_(y) <==> x/y

**\_\_divmod\_\_**(...)  
x. \_\_divmod\_\_(y) <==> divmod(x, y)

**\_\_float\_\_**(...)  
x. \_\_float\_\_() <==> float(x)

**\_\_floordiv\_\_**(...)  
x. \_\_floordiv\_\_(y) <==> x//y

```
format(...)
 NumPy array scalar formatter

getitem(...)
 x.getitem(y) <==> x[y]

hex(...)
 x.hex() <==> hex(x)

int(...)
 x.int() <==> int(x)

invert(...)
 x.invert() <==> ~x

long(...)
 x.long() <==> long(x)

lshift(...)
 x.lshift(y) <==> x<<y

mod(...)
 x.mod(y) <==> x%y

mul(...)
 x.mul(y) <==> x*y

neg(...)
 x.neg() <==> -x

nonzero(...)
 x.nonzero() <==> x != 0

oct(...)
 x.oct() <==> oct(x)

or(...)
 x.or(y) <==> x|y

pos(...)
 x.pos() <==> +x

pow(...)
 x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
 x.radd(y) <==> y+x

rand(...)
 x.rand(y) <==> y&x

rdiv(...)
 x.rdiv(y) <==> y/x

rdivmod(...)
 x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
 x.repr() <==> repr(x)

rfloordiv(...)
 x.rfloordiv(y) <==> y//x

rlshift(...)
 x.rlshift(y) <==> y<<x

rmod(...)
 x.rmod(y) <==> y%x

rmul(...)
 x.rmul(y) <==> y*x
```

```
__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----  
new\_order : [str](#), optional  
Byte order to force; a value from the [byte](#) order specifications

above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----  
`new_dtype : dtype`  
New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
resize(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
round(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
searchsorted(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
setfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
setflags(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
sort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
squeeze(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
std(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
```

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

**\_\_array\_interface\_\_**

Array protocol: Python side

**\_\_array\_priority\_\_**

Array priority.

**\_\_array\_struct\_\_**

Array protocol: struct

**base**

base object

**data**

pointer to start of data

**dtype**

get array data-descriptor

**flags**

integer value of flags

**flat**

a 1-d view of scalar

**imag**

imaginary part of scalar

**itemsize**

length of one element in bytes

**nbytes**

length of item in bytes

**ndim**

number of array dimensions

**real**

```

 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

class int32(signedinteger)
32-bit integer. Character code 'i'. C int compatible.

```

Method resolution order:

```

int32
signedinteger
integer
number
generic
builtin object

```

---

Methods defined here:

```

eq(...)
 x. _eq_(y) <==> x==y

ge(...)
 x. _ge_(y) <==> x>=y

gt(...)
 x. _gt_(y) <==> x>y

hash(...)
 x. _hash_() <==> hash(x)

index(...)
 x[y:z] <==> x[y. _index_():z. _index_()]

le(...)
 x. _le_(y) <==> x<=y

lt(...)
 x. _lt_(y) <==> x<y

ne(...)
 x. _ne_(y) <==> x!=y

```

---

Data and other attributes defined here:

```

new = <built-in method _new_ of type object>
 T. _new_(S, ...) -> a new object with type S, a subtype of T

```

---

Data descriptors inherited from [integer](#):

```

denominator
 denominator of value (1)

numerator
 numerator of value (the value itself)

```

---

Methods inherited from [generic](#):

```

abs(...)
 x. _abs_() <==> abs(x)

add(...)
 x. _add_(y) <==> x+y

```

```
__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
```

```
x.rand(y) <==> y&x
rdiv_(...)
x.rdiv(y) <==> y/x
rdivmod_(...)
x.rdivmod(y) <==> divmod(y, x)
reduce_(...)
repr_(...)
x.repr() <==> repr(x)
rfloordiv_(...)
x.rfloordiv(y) <==> y//x
rlshift_(...)
x.rlshift(y) <==> y<<x
rmod_(...)
x.rmod(y) <==> y%x
rmul_(...)
x.rmul(y) <==> y*x
ror_(...)
x.ror(y) <==> y|x
rpow_(...)
y.rpow(x[, z]) <==> pow(x, y[, z])
rrshift_(...)
x.rrshift(y) <==> y>>x
rshift_(...)
x.rshift(y) <==> x>>y
rsub_(...)
x.rsub(y) <==> y-x
rtruediv_(...)
x.rtruediv(y) <==> y/x
rxor_(...)
x.rxor(y) <==> y^x
setstate_(...)
sizeof_(...)
str_(...)
x.str() <==> str(x)
sub_(...)
x.sub(y) <==> x-y
truediv_(...)
x.truediv(y) <==> x/y
xor_(...)
x.xor(y) <==> x^y
all(...)
Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str`, optional

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**put(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

class **int64**([signedinteger](#), [\\_builtin\\_.int](#))  
64-bit [integer](#). Character code 'l'. Python [int](#) compatible.

Method resolution order:

[int64](#)  
[signedinteger](#)  
[integer](#)  
[number](#)  
[generic](#)  
[\\_builtin\\_.int](#)  
[\\_builtin\\_.object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_\(...\)](#)  
x. [\\_\\_eq\\_\\_\(y\)](#) <==> x==y

[\\_\\_ge\\_\\_\(...\)](#)  
x. [\\_\\_ge\\_\\_\(y\)](#) <==> x>=y

[\\_\\_gt\\_\\_\(...\)](#)  
x. [\\_\\_gt\\_\\_\(y\)](#) <==> x>y

[\\_\\_index\\_\\_\(...\)](#)  
x[y:z] <==> x[y. [\\_\\_index\\_\\_\(\)](#):z. [\\_\\_index\\_\\_\(\)](#)]

[\\_\\_le\\_\\_\(...\)](#)  
x. [\\_\\_le\\_\\_\(y\)](#) <==> x<=y

[\\_\\_lt\\_\\_\(...\)](#)  
x. [\\_\\_lt\\_\\_\(y\)](#) <==> x<y

[\\_\\_ne\\_\\_\(...\)](#)  
x. [\\_\\_ne\\_\\_\(y\)](#) <==> x!=y

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T.**\_\_new\_\_**(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x.**\_\_abs\_\_**() <==> abs(x)

**\_\_add\_\_(...)**  
x.**\_\_add\_\_**(y) <==> x+y

**\_\_and\_\_(...)**  
x.**\_\_and\_\_**(y) <==> x&y

**\_\_array\_\_(...)**  
sc.**\_\_array\_\_**(|type) return 0-dim array

**\_\_array\_wrap\_\_(...)**  
sc.**\_\_array\_wrap\_\_**(obj) return scalar from array

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**  
x.**\_\_div\_\_**(y) <==> x/y

**\_\_divmod\_\_(...)**  
x.**\_\_divmod\_\_**(y) <==> divmod(x, y)

**\_\_float\_\_(...)**  
x.**\_\_float\_\_**() <==> float(x)

**\_\_floordiv\_\_(...)**  
x.**\_\_floordiv\_\_**(y) <==> x//y

**\_\_format\_\_(...)**  
NumPy array scalar formatter

**\_\_getitem\_\_(...)**  
x.**\_\_getitem\_\_**(y) <==> x[y]

**\_\_hex\_\_(...)**  
x.**\_\_hex\_\_**() <==> hex(x)

**\_\_int\_\_(...)**  
x.**\_\_int\_\_**() <==> int(x)

**\_\_invert\_\_(...)**  
x.**\_\_invert\_\_**() <==> ~x

**\_\_long\_\_(...)**  
x.**\_\_long\_\_**() <==> long(x)

**\_\_lshift\_\_(...)**  
x.**\_\_lshift\_\_**(y) <==> x<<y

**\_\_mod\_\_(...)**  
x.**\_\_mod\_\_**(y) <==> x%y

**\_\_mul\_\_(...)**  
x.**\_\_mul\_\_**(y) <==> x\*y

**\_\_neg\_\_(...)**  
x.**\_\_neg\_\_**() <==> -x

```
nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y
```

```
truediv(...)
x._truediv_(y) <==> x/y

xor(...)
x._xor_(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New [dtype](#) object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
tobytes(...)
tofile(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tolist(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

toString(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

Methods inherited from [\\_\\_builtin\\_\\_.int](#):

**\_\_cmp\_\_(...)**  
    x. [\\_\\_cmp\\_\\_](#)(y) <==> cmp(x,y)

**\_\_coerce\_\_(...)**  
    x. [\\_\\_coerce\\_\\_](#)(y) <==> coerce(x, y)

**\_\_getattribute\_\_(...)**  
    x. [\\_\\_getattribute\\_\\_](#)('name') <==> x.name

**\_\_getnewargs\_\_(...)**

**\_\_hash\_\_(...)**  
    x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

**\_\_trunc\_\_(...)**  
    Truncating an Integral returns itself.

**bit\_length(...)**  
    [int.bit\\_length\(\)](#) -> [int](#)

    Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

---

class **int8**([signedinteger](#))  
8-bit [integer](#). Character code ``b''. C char compatible.

Method resolution order:

```
int8
signedinteger
integer
number
generic
_builtin_object
```

---

Methods defined here:

```
__eq__\(...\)
x. __eq__\(y\) <==> x==y

__ge__\(...\)
x. __ge__\(y\) <==> x>=y

__gt__\(...\)
x. __gt__\(y\) <==> x>y

__hash__\(...\)
x. __hash__\(\) <==> hash(x)

__index__\(...\)
x[y:z] <==> x[y. __index__\(\):z. __index__\(\)]

__le__\(...\)
x. __le__\(y\) <==> x<=y

__lt__\(...\)
x. __lt__\(y\) <==> x<y

__ne__\(...\)
x. __ne__\(y\) <==> x!=y
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T. __new__\(S, ...\) -> a new object with type S, a subtype of T
```

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

```
__abs__\(...\)
x. __abs__\(\) <==> abs(x)

__add__\(...\)
x. __add__\(y\) <==> x+y

__and__\(...\)
x. __and__\(y\) <==> x&y

__array__\(...\)
sc. __array__\(|type\) return 0-dim array
```

```
_array_wrap_(...)
 sc.array_wrap(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
 x.div(y) <==> x/y

divmod(...)
 x.divmod(y) <==> divmod(x, y)

float(...)
 x.float() <==> float(x)

floordiv(...)
 x.floordiv(y) <==> x//y

format(...)
 NumPy array scalar formatter

getitem(...)
 x.getitem(y) <==> x[y]

hex(...)
 x.hex() <==> hex(x)

int(...)
 x.int() <==> int(x)

invert(...)
 x.invert() <==> ~x

long(...)
 x.long() <==> long(x)

lshift(...)
 x.lshift(y) <==> x<<y

mod(...)
 x.mod(y) <==> x%y

mul(...)
 x.mul(y) <==> x*y

neg(...)
 x.neg() <==> -x

nonzero(...)
 x.nonzero() <==> x != 0

oct(...)
 x.oct() <==> oct(x)

or(...)
 x.or(y) <==> x|y

pos(...)
 x.pos() <==> +x

pow(...)
 x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
 x.radd(y) <==> y+x

rand(...)
 x.rand(y) <==> y&x

rdiv(...)
 x.rdiv(y) <==> y/x

rdivmod(...)
```

```
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y

truediv(...)
x.truediv(y) <==> x/y

xor(...)
x.xor(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
conj(...)
conjugate(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

copy(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

cumprod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

cumsum(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

diagonal(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

dump(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

dumps(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

fill(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

```
newbyteorder(...)
 newbyteorder\(new_order='S'\)

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 Parameters

 new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**taked(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**

```

 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

```

---

**int\_** = class int64([signedinteger](#), [builtin\\_int](#))  
 64-bit [integer](#). Character code 'l'. Python [int](#) compatible.

Method resolution order:

```

int64
signedinteger
integer
number
generic
builtin_int
builtin_object

```

---

Methods defined here:

```

eq(...)
 x. _eq_(y) <==> x==y

ge(...)
 x. _ge_(y) <==> x>=y

gt(...)
 x. _gt_(y) <==> x>y

index(...)
 x[y:z] <==> x[y. _index_():z. _index_()]

le(...)
 x. _le_(y) <==> x<=y

lt(...)
 x. _lt_(y) <==> x<y

ne(...)
 x. _ne_(y) <==> x!=y

```

---

Data and other attributes defined here:

```

new = <built-in method __new__ of type object>
 T. _new_(S, ...) -> a new object with type S, a subtype of T

```

---

Data descriptors inherited from [integer](#):

**denominator**

```
denominator of value (1)

numerator
 numerator of value (the value itself)

Methods inherited from generic:

abs(...)
x._abs_() <==> abs(x)

add(...)
x._add_(y) <==> x+y

and(...)
x._and_(y) <==> x&y

array(...)
sc._array_(|type) return 0-dim array

_array_wrap_(...)
sc._array_wrap_(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
x._div_(y) <==> x/y

divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)


```

```
__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y
```

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----

`new_dtype : dtype`  
New `dtype` [object](#) with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_array\_interface\_**  
    Array protocol: Python side

```
__array_priority__
 Array priority.

__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

Methods inherited from __builtin__.int:

__cmp__(...)
x.__cmp__(y) <==> cmp(x,y)

__coerce__(...)
x.__coerce__(y) <==> coerce(x, y)

__getattribute__(...)
x.__getattribute__('name') <==> x.name

__getnewargs__(...)

__hash__(...)
x.__hash__() <==> hash(x)

__trunc__(...)
Truncating an Integral returns itself.

bit_length(...)
int.bit_length\(\) -> int

Number of bits necessary to represent self in binary.
>>> bin(37)
'0b100101'
>>> (37).bit_length\(\)
6


```

---

```
intc = class int32(signedinteger)
32-bit integer. Character code 'i'. C int compatible.
```

Method resolution order:

```
int32
signedinteger
integer
number
generic
builtin .object
```

---

Methods defined here:

```
__eq__(...)
x. __eq__(y) <==> x==y

__ge__(...)
x. __ge__(y) <==> x>=y

__gt__(...)
x. __gt__(y) <==> x>y

__hash__(...)
x. __hash__() <==> hash(x)

__index__(...)
x[y:z] <==> x[y. __index__():z. __index__()]

__le__(...)
x. __le__(y) <==> x<=y

__lt__(...)
x. __lt__(y) <==> x<y

__ne__(...)
x. __ne__(y) <==> x!=y
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T. __new__(S, ...) -> a new object with type S, a subtype of T
```

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

```
__abs__(...)
x. __abs__() <==> abs(x)

__add__(...)
x. __add__(y) <==> x+y

__and__(...)
x. __and__(y) <==> x&y

__array__(...)
sc. __array__(|type) return 0-dim array

__array_wrap__(...)
sc. __array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)
```

```
div(...)
x._div_(y) <==> x/y

divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

repr(...)
x._repr_() <==> repr(x)
```

```
__rfloordiv__(...)
 x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
 x. __rlshift__(y) <==> y<<x

__rmod__(...)
 x. __rmod__(y) <==> y%x

__rmul__(...)
 x. __rmul__(y) <==> y*x

__ror__(...)
 x. __ror__(y) <==> y|x

__rpow__(...)
 y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x. __rrshift__(y) <==> y>>x

__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `[dtype](#)` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New [dtype](#) object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

**class integer(number)**

Method resolution order:

[integer](#)  
[number](#)  
[generic](#)  
[builtin\\_object](#)

---

Data descriptors defined here:

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x. [\\_\\_abs\\_\\_](#)() <==> abs(x)

**\_\_add\_\_(...)**  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y

**\_\_and\_\_(...)**  
x. [\\_\\_and\\_\\_](#)(y) <==> x&y

**\_\_array\_\_(...)**  
sc. [\\_\\_array\\_\\_](#)(|type) return 0-dim array

**\_\_array\_wrap\_\_(...)**  
sc. [\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**  
x. [\\_\\_div\\_\\_](#)(y) <==> x/y

**\_\_divmod\_\_(...)**  
x. [\\_\\_divmod\\_\\_](#)(y) <==> divmod(x, y)

**\_\_eq\_\_(...)**  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

**\_\_float\_\_(...)**  
x. [\\_\\_float\\_\\_](#)() <==> [float](#)(x)

**\_\_floordiv\_\_(...)**

```
x. floordiv_(y) <==> x//y
format_(...)
 NumPy array scalar formatter
ge_(...)
 x. ge_(y) <==> x>=y
getitem_(...)
 x. getitem_(y) <==> x[y]
gt_(...)
 x. gt_(y) <==> x>y
hex_(...)
 x. hex_() <==> hex(x)
int_(...)
 x. int_() <==> int(x)
invert_(...)
 x. invert_() <==> ~x
le_(...)
 x. le_(y) <==> x<=y
long_(...)
 x. long_() <==> long(x)
lshift_(...)
 x. lshift_(y) <==> x<<y
lt_(...)
 x. lt_(y) <==> x<y
mod_(...)
 x. mod_(y) <==> x%y
mul_(...)
 x. mul_(y) <==> x*y
ne_(...)
 x. ne_(y) <==> x!=y
neg_(...)
 x. neg_() <==> -x
nonzero_(...)
 x. nonzero_() <==> x != 0
oct_(...)
 x. oct_() <==> oct(x)
or_(...)
 x. or_(y) <==> x|y
pos_(...)
 x. pos_() <==> +x
pow_(...)
 x. pow_(y[, z]) <==> pow(x, y[, z])
radd_(...)
 x. radd_(y) <==> y+x
rand_(...)
 x. rand_(y) <==> y&x
rdiv_(...)
 x. rdiv_(y) <==> y/x
rdivmod_(...)
 x. rdivmod_(y) <==> divmod(y, x)
```

```
__reduce__(...)

__repr__(...)
 x.__repr__() <==> repr(x)

__rfloordiv__(...)
 x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
 x.__rlshift__(y) <==> y<<x

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(y) <==> y*x

__ror__(...)
 x.__ror__(y) <==> y|x

__rpow__(...)
 y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x.__rrshift__(y) <==> y>>x

__rshift__(...)
 x.__rshift__(y) <==> x>>y

__rsub__(...)
 x.__rsub__(y) <==> y-x

__rtruediv__(...)
 x.__rtruediv__(y) <==> y/x

__rxor__(...)
 x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x.__str__() <==> str(x)

__sub__(...)
 x.__sub__(y) <==> x-y

__truediv__(...)
 x.__truediv__(y) <==> x/y

__xor__(...)
 x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

```
newbyteorder(...)
 newbyteorder\(new_order='S'\)

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 Parameters

 new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
squeeze(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

taked(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**

```

 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

```

---

**intp** = class int64([signedinteger](#), [builtin\\_int](#))  
 64-bit [integer](#). Character code 'l'. Python [int](#) compatible.

Method resolution order:

```

int64
signedinteger
integer
number
generic
builtin_int
builtin_object

```

---

Methods defined here:

```

__eq__(...)
 x. __eq__(y) <==> x==y

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__index__(...)
 x[y:z] <==> x[y. __index__():z. __index__()]

__le__(...)
 x. __le__(y) <==> x<=y

__lt__(...)
 x. __lt__(y) <==> x<y

__ne__(...)
 x. __ne__(y) <==> x!=y

```

---

Data and other attributes defined here:

```

__new__ = <built-in method __new__ of type object>
 T. __new__(S, ...) -> a new object with type S, a subtype of T

```

---

Data descriptors inherited from [integer](#):

**denominator**

```
denominator of value (1)

numerator
 numerator of value (the value itself)

Methods inherited from generic:

abs(...)
x._abs_() <==> abs(x)

add(...)
x._add_(y) <==> x+y

and(...)
x._and_(y) <==> x&y

array(...)
sc._array_(|type) return 0-dim array

_array_wrap_(...)
sc._array_wrap_(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
x._div_(y) <==> x/y

divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)


```

```
__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y
```

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

clip(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

compress(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

conj(...)

conjugate(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

copy(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumprod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

cumsum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

diagonal(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

dump(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----

`new_dtype : dtype`  
New `dtype` [object](#) with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_array\_interface\_**  
    Array protocol: Python side

---

**\_\_array\_priority\_\_**  
Array priority.

**\_\_array\_struct\_\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

Methods inherited from [builtin\\_int](#):

**\_\_cmp\_\_(...)**  
`x.cmp(y) <==> cmp(x,y)`

**\_\_coerce\_\_(...)**  
`x.coerce(y) <==> coerce(x, y)`

**\_\_getattribute\_\_(...)**  
`x.getattribute('name') <==> x.name`

**\_\_getnewargs\_\_(...)**

**\_\_hash\_\_(...)**  
`x.hash() <==> hash(x)`

**\_\_trunc\_\_(...)**  
Truncating an Integral returns itself.

**bit\_length(...)**  
`int.bit_length() -> int`

Number of bits necessary to represent self in binary.  
`>>> bin(37)  
'0b100101'  
>>> (37).bit_length()  
6`

**longcomplex** = class complex256([complexfloating](#))

Composed of two 128 bit floats

Method resolution order:

```
complex256
complexfloating
inexact
number
generic
builtin.object
```

---

Methods defined here:

```
__complex__\(...\)
__eq__\(...\)
 x. __eq__\(y\) <==> x==y
__float__\(...\)
 x. __float__\(\) <==> float\(x\)
__ge__\(...\)
 x. __ge__\(y\) <==> x>=y
__gt__\(...\)
 x. __gt__\(y\) <==> x>y
__hash__\(...\)
 x. __hash__\(\) <==> hash(x)
__hex__\(...\)
 x. __hex__\(\) <==> hex(x)
__int__\(...\)
 x. __int__\(\) <==> int\(x\)
__le__\(...\)
 x. __le__\(y\) <==> x<=y
__long__\(...\)
 x. __long__\(\) <==> long(x)
__lt__\(...\)
 x. __lt__\(y\) <==> x<y
__ne__\(...\)
 x. __ne__\(y\) <==> x!=y
__oct__\(...\)
 x. __oct__\(\) <==> oct(x)
__repr__\(...\)
 x. __repr__\(\) <==> repr(x)
__str__\(...\)
 x. __str__\(\) <==> str\(x\)
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T. __new__\(S, ...\) -> a new object with type S, a subtype of T
```

---

Methods inherited from [generic](#):

```
__abs__\(...\)
 x. __abs__\(\) <==> abs(x)
__add__\(...\)
 x. __add__\(y\) <==> x+y
```

```
__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__invert__(...)
x.__invert__() <==> ~x

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x
```

**\_\_rmod\_\_(...)**  
x. rmod(y) <==> y%x

**\_\_rmul\_\_(...)**  
x. rmul(y) <==> y\*x

**\_\_ror\_\_(...)**  
x. ror(y) <==> y|x

**\_\_rpow\_\_(...)**  
y. rpow(x[, z]) <==> pow(x, y[, z])

**\_\_rrshift\_\_(...)**  
x. rrshift(y) <==> y>>x

**\_\_rshift\_\_(...)**  
x. rshift(y) <==> x>>y

**\_\_rsub\_\_(...)**  
x. rsub(y) <==> y-x

**\_\_rtruediv\_\_(...)**  
x. rtruediv(y) <==> y/x

**\_\_rxor\_\_(...)**  
x. rxor(y) <==> y^x

**\_\_setstate\_\_(...)**

**\_\_sizeof\_\_(...)**

**\_\_sub\_\_(...)**  
x. sub(y) <==> x-y

**\_\_truediv\_\_(...)**  
x. truediv(y) <==> x/y

**\_\_xor\_\_(...)**  
x. xor(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

```
* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)
```

Parameters

```

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

>Returns

new_dtype : dtype
 New 'dtype' object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

**longdouble** = class float128([floating](#))  
128-bit [floating](#)-point [number](#). Character code: 'g'. C long [float](#) compatible.

Method resolution order:

[float128](#)  
[floating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[builtin](#) [object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_](#)(...)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_float\\_\\_](#)(...)  
x. [\\_\\_float\\_\\_](#)() <==> [float](#)(x)

[\\_\\_ge\\_\\_](#)(...)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_](#)(...)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_](#)(...)  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_hex\\_\\_](#)(...)  
x. [\\_\\_hex\\_\\_](#)() <==> hex(x)

[\\_\\_int\\_\\_](#)(...)  
x. [\\_\\_int\\_\\_](#)() <==> [int](#)(x)

[\\_\\_le\\_\\_](#)(...)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_long\\_\\_](#)(...)  
x. [\\_\\_long\\_\\_](#)() <==> [long](#)(x)

[\\_\\_lt\\_\\_](#)(...)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_](#)(...)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

[\\_\\_oct\\_\\_](#)(...)  
x. [\\_\\_oct\\_\\_](#)() <==> oct(x)

[\\_\\_repr\\_\\_](#)(...)  
x. [\\_\\_repr\\_\\_](#)() <==> [repr](#)(x)

[\\_\\_str\\_\\_](#)(...)  
x. [\\_\\_str\\_\\_](#)() <==> [str](#)(x)

---

Data and other attributes defined here:

`__new__ = <built-in method __new__ of type object>`  
`T.__new__(S, ...) -> a new object with type S, a subtype of T`

---

Methods inherited from [generic](#):

`__abs__(...)`  
`x.__abs__() <==> abs(x)`

`__add__(...)`  
`x.__add__(y) <==> x+y`

`__and__(...)`  
`x.__and__(y) <==> x&y`

`__array__(...)`  
`sc.__array__(|type) return 0-dim array`

`__array_wrap__(...)`  
`sc.__array_wrap__(obj) return scalar from array`

`__copy__(...)`

`__deepcopy__(...)`

`__div__(...)`  
`x.__div__(y) <==> x/y`

`__divmod__(...)`  
`x.__divmod__(y) <==> divmod(x, y)`

`__floordiv__(...)`  
`x.__floordiv__(y) <==> x//y`

`__format__(...)`  
`NumPy array scalar formatter`

`__getitem__(...)`  
`x.__getitem__(y) <==> x[y]`

`__invert__(...)`  
`x.__invert__() <==> ~x`

`__lshift__(...)`  
`x.__lshift__(y) <==> x<<y`

`__mod__(...)`  
`x.__mod__(y) <==> x%y`

`__mul__(...)`  
`x.__mul__(y) <==> x*y`

`__neg__(...)`  
`x.__neg__() <==> -x`

`__nonzero__(...)`  
`x.__nonzero__() <==> x != 0`

`__or__(...)`  
`x.__or__(y) <==> x|y`

`__pos__(...)`  
`x.__pos__() <==> +x`

`__pow__(...)`  
`x.__pow__(y[, z]) <==> pow(x, y[, z])`

`__radd__(...)`  
`x.__radd__(y) <==> y+x`

`__rand__(...)`  
`x.__rand__(y) <==> y&x`

`__rdiv__(...)`

```
x.__rdiv__(y) <==> y/x
__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)
__reduce__(...)
__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x
__rlshift__(...)
x.__rlshift__(y) <==> y<<x
__rmod__(...)
x.__rmod__(y) <==> y%x
__rmul__(...)
x.__rmul__(y) <==> y*x
__ror__(...)
x.__ror__(y) <==> y|x
__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])
__rrshift__(...)
x.__rrshift__(y) <==> y>>x
__rshift__(...)
x.__rshift__(y) <==> x>>y
__rsub__(...)
x.__rsub__(y) <==> y-x
__rtruediv__(...)
x.__rtruediv__(y) <==> y/x
__rxor__(...)
x.__rxor__(y) <==> y^x
__setstate__(...)
__sizeof__(...)
__sub__(...)
x.__sub__(y) <==> x-y
__truediv__(...)
x.__truediv__(y) <==> x/y
__xor__(...)
x.__xor__(y) <==> x^y
all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

```
newbyteorder(new_order='S')

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

repeat(...)

 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

**longfloat** = class float128([floating](#))  
128-bit [floating](#)-point [number](#). Character code: 'g'. C long [float](#)  
compatible.

Method resolution order:

[float128](#)  
[floating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[builtin](#) .[object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_\(...\)](#)  
    x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_float\\_\\_\(...\)](#)  
    x. [\\_\\_float\\_\\_](#)() <==> [float](#)(x)

[\\_\\_ge\\_\\_\(...\)](#)  
    x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_\(...\)](#)  
    x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_\(...\)](#)  
    x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_hex\\_\\_\(...\)](#)  
    x. [\\_\\_hex\\_\\_](#)() <==> hex(x)

[\\_\\_int\\_\\_\(...\)](#)  
    x. [\\_\\_int\\_\\_](#)() <==> [int](#)(x)

[\\_\\_le\\_\\_\(...\)](#)  
    x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_long\\_\\_\(...\)](#)  
    x. [\\_\\_long\\_\\_](#)() <==> [long](#)(x)

[\\_\\_lt\\_\\_\(...\)](#)  
    x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_\(...\)](#)  
    x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

---

```
__oct__(...)
 x.__oct__() <==> oct(x)

__repr__(...)
 x.__repr__() <==> repr(x)

__str__(...)
 x.__str__() <==> str(x)
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

---

Methods inherited from [generic](#):

```
__abs__(...)
 x.__abs__() <==> abs(x)

__add__(...)
 x.__add__(y) <==> x+y

__and__(...)
 x.__and__(y) <==> x&y

__array__(...)
 sc.__array__(|type) return 0-dim array

__array_wrap__(...)
 sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x.__div__(y) <==> x/y

__divmod__(...)
 x.__divmod__(y) <==> divmod(x, y)

__floordiv__(...)
 x.__floordiv__(y) <==> x//y

__format__(...)
 NumPy array scalar formatter

__getitem__(...)
 x.__getitem__(y) <==> x[y]

__invert__(...)
 x.__invert__() <==> ~x

__lshift__(...)
 x.__lshift__(y) <==> x<<y

__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(y) <==> x*y

__neg__(...)
 x.__neg__() <==> -x

__nonzero__(...)
 x.__nonzero__() <==> x != 0

__or__(...)
 x.__or__(y) <==> x|y

__pos__(...)
 x.__pos__() <==> +x
```

**\_\_pow\_\_**(...)  
x.\_\_pow\_\_(y[, z]) <==> pow(x, y[, z])

**\_\_radd\_\_**(...)  
x.\_\_radd\_\_(y) <==> y+x

**\_\_rand\_\_**(...)  
x.\_\_rand\_\_(y) <==> y&x

**\_\_rdiv\_\_**(...)  
x.\_\_rdiv\_\_(y) <==> y/x

**\_\_rdivmod\_\_**(...)  
x.\_\_rdivmod\_\_(y) <==> divmod(y, x)

**\_\_reduce\_\_**(...)

**\_\_rfloordiv\_\_**(...)  
x.\_\_rfloordiv\_\_(y) <==> y//x

**\_\_rlshift\_\_**(...)  
x.\_\_rlshift\_\_(y) <==> y<<x

**\_\_rmod\_\_**(...)  
x.\_\_rmod\_\_(y) <==> y%x

**\_\_rmul\_\_**(...)  
x.\_\_rmul\_\_(y) <==> y\*x

**\_\_ror\_\_**(...)  
x.\_\_ror\_\_(y) <==> y|x

**\_\_rpow\_\_**(...)  
y.\_\_rpow\_\_(x[, z]) <==> pow(x, y[, z])

**\_\_rrshift\_\_**(...)  
x.\_\_rrshift\_\_(y) <==> y>>x

**\_\_rshift\_\_**(...)  
x.\_\_rshift\_\_(y) <==> x>>y

**\_\_rsub\_\_**(...)  
x.\_\_rsub\_\_(y) <==> y-x

**\_\_rtruediv\_\_**(...)  
x.\_\_rtruediv\_\_(y) <==> y/x

**\_\_rxor\_\_**(...)  
x.\_\_rxor\_\_(y) <==> y^x

**\_\_setstate\_\_**(...)

**\_\_sizeof\_\_**(...)

**\_\_sub\_\_**(...)  
x.\_\_sub\_\_(y) <==> x-y

**\_\_truediv\_\_**(...)  
x.\_\_truediv\_\_(y) <==> x/y

**\_\_xor\_\_**(...)  
x.\_\_xor\_\_(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)

Class **generic** exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the **ndarray** class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmax(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

```
min(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder\(new_order='S'\)

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 Parameters

 new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **tostring(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

## Data descriptors inherited from [generic](#):

### **T**

transpose

### **\_\_array\_interface\_\_**

Array protocol: Python side

### **\_\_array\_priority\_\_**

Array priority.

### **\_\_array\_struct\_\_**

Array protocol: struct

### **base**

base object

```

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

```

---

```

longlong = class int64(signedinteger, builtin_.int)
Method resolution order:
 int64
 signedinteger
 integer
 number
 generic
 builtin_.int
 builtin_.object

```

---

Methods defined here:

```

__eq__(...)
 x. __eq__(y) <==> x==y

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__index__(...)
 x[y:z] <==> x[y. __index__() : z. __index__()]

__le__(...)
 x. __le__(y) <==> x<=y

__lt__(...)
 x. __lt__(y) <==> x<y

__ne__(...)
 x. __ne__(y) <==> x!=y

```

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T.**\_\_new\_\_**(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x.**\_\_abs\_\_**() <==> abs(x)

**\_\_add\_\_(...)**  
x.**\_\_add\_\_**(y) <==> x+y

**\_\_and\_\_(...)**  
x.**\_\_and\_\_**(y) <==> x&y

**\_\_array\_\_(...)**  
sc.**\_\_array\_\_**(|type) return 0-dim array

**\_\_array\_wrap\_\_(...)**  
sc.**\_\_array\_wrap\_\_**(obj) return scalar from array

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**  
x.**\_\_div\_\_**(y) <==> x/y

**\_\_divmod\_\_(...)**  
x.**\_\_divmod\_\_**(y) <==> divmod(x, y)

**\_\_float\_\_(...)**  
x.**\_\_float\_\_**() <==> float(x)

**\_\_floordiv\_\_(...)**  
x.**\_\_floordiv\_\_**(y) <==> x//y

**\_\_format\_\_(...)**  
NumPy array scalar formatter

**\_\_getitem\_\_(...)**  
x.**\_\_getitem\_\_**(y) <==> x[y]

**\_\_hex\_\_(...)**  
x.**\_\_hex\_\_**() <==> hex(x)

**\_\_int\_\_(...)**  
x.**\_\_int\_\_**() <==> int(x)

**\_\_invert\_\_(...)**  
x.**\_\_invert\_\_**() <==> ~x

**\_\_long\_\_(...)**  
x.**\_\_long\_\_**() <==> long(x)

**\_\_lshift\_\_(...)**  
x.**\_\_lshift\_\_**(y) <==> x<<y

**\_\_mod\_\_(...)**  
x.**\_\_mod\_\_**(y) <==> x%y

**\_\_mul\_\_(...)**  
x.**\_\_mul\_\_**(y) <==> x\*y

**\_\_neg\_\_(...)**  
x.**\_\_neg\_\_**() <==> -x

```
nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y
```

```
truediv(...)
x._truediv_(y) <==> x/y

xor(...)
x._xor_(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New [dtype](#) object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
tobytes(...)
tofile(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tolist(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

toString(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

Methods inherited from [\\_\\_builtin\\_\\_.int](#):

**\_\_cmp\_\_(...)**  
    x. [\\_\\_cmp\\_\\_](#)(y) <==> cmp(x,y)

**\_\_coerce\_\_(...)**  
    x. [\\_\\_coerce\\_\\_](#)(y) <==> coerce(x, y)

**\_\_getattribute\_\_(...)**  
    x. [\\_\\_getattribute\\_\\_](#)('name') <==> x.name

**\_\_getnewargs\_\_(...)**

**\_\_hash\_\_(...)**  
    x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

**\_\_trunc\_\_(...)**  
    Truncating an Integral returns itself.

**bit\_length(...)**  
    [int.bit\\_length\(\)](#) -> [int](#)

        Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6

class matrix(numpy.ndarray)
 matrix(data, dtype=None, copy=True)

 Returns a matrix from an array-like object, or from a string of data.
 A matrix is a specialized 2-D array that retains its 2-D nature
 through operations. It has certain special operators, such as ``*``
 (matrix multiplication) and ``**`` (matrix power).

 Parameters

 data : array_like or string
 If `data` is a string, it is interpreted as a matrix with commas
 or spaces separating columns, and semicolons separating rows.
 dtype : data-type
 Data-type of the output matrix.
 copy : bool
 If `data` is already an ndarray, then this flag determines
 whether the data is copied (the default), or whether a view is
 constructed.

 See Also

 array

 Examples

 >>> a = np.matrix('1 2; 3 4')
 >>> print(a)
 [[1 2]
 [3 4]]

 >>> np.matrix([[1, 2], [3, 4]])
 matrix([[1, 2],
 [3, 4]])

 Method resolution order:
 matrix
 numpy.ndarray
 builtin_object

Methods defined here:

__array_finalize__\(self, obj\)
__getitem__\(self, index\)
__imul__\(self, other\)
__ipow__\(self, other\)
__mul__\(self, other\)
__pow__\(self, other\)
__repr__\(self\)
__rmul__\(self, other\)
__rpow__\(self, other\)
__str__\(self\)

all(self, axis=None, out=None)
Test whether all matrix elements along a given axis evaluate to True.

Parameters

See `numpy.all` for complete descriptions

See Also

numpy.all

Notes


```

```

This is the same as `ndarray.all`, but it returns a `matrix` object.

Examples

>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> y = x[0]; y
matrix([[0, 1, 2, 3]])
>>> (x == y)
matrix([[True, True, True, True],
 [False, False, False, False],
 [False, False, False, False]], dtype=bool)
>>> (x == y).all()
False
>>> (x == y).all(0)
matrix([[False, False, False, False]], dtype=bool)
>>> (x == y).all(1)
matrix([[True],
 [False],
 [False]], dtype=bool)

any(self, axis=None, out=None)
Test whether any array element along a given axis evaluates to True.

Refer to `numpy.any` for full documentation.

Parameters

axis : int, optional
 Axis along which logical OR is performed
out : ndarray, optional
 Output to existing array instead of creating new one, must have
 same shape as expected output

Returns

any : bool, ndarray
 Returns a single bool if `axis` is ``None``; otherwise,
 returns `ndarray`
```

**argmax(self, axis=None, out=None)**

Indexes of the maximum values along an axis.

Return the indexes of the first occurrences of the maximum values along the specified axis. If axis is None, the index is for the flattened matrix.

Parameters

See `numpy.argmax` for complete descriptions

See Also

numpy.argmax

Notes

This is the same as `ndarray.argmax`, but returns a `matrix` object where `ndarray.argmax` would return an `ndarray`.

Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.argmax()
11
>>> x.argmax(0)
matrix([[2, 2, 2, 2]])
>>> x.argmax(1)
matrix([[3],
 [3],
 [3]])
```

**argmin(self, axis=None, out=None)**

Indexes of the minimum values along an axis.

Return the indexes of the first occurrences of the minimum values along the specified axis. If axis is None, the index is for the flattened matrix.

Parameters

See `numpy.argmin` for complete descriptions.

See Also  
-----  
[numpy.argmax](#)

Notes  
-----  
This is the same as `ndarray.argmax`, but returns a `matrix` object where `ndarray.argmax` would return an `ndarray`.

Examples  
-----  

```
>>> x = -np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, -1, -2, -3],
 [-4, -5, -6, -7],
 [-8, -9, -10, -11]])
>>> x.argmax()
11
>>> x.argmax(0)
matrix([[2, 2, 2, 2]])
>>> x.argmax(1)
matrix([3],
 [3],
 [3]))
```

**flatten(self, order='C')**  
Return a flattened copy of the [matrix](#).  
All 'N' elements of the [matrix](#) are placed into a [single](#) row.  
Parameters  
-----  
order : {'C', 'F', 'A', 'K'}, optional  
'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran-style) order. 'A' means to flatten in column-major order if 'm' is Fortran \*contiguous\* in memory, row-major order otherwise. 'K' means to flatten 'm' in the order the elements occur in memory. The default is 'C'.  
Returns  
-----  
y : [matrix](#)  
A copy of the [matrix](#), flattened to a `(1, N)` [matrix](#) where 'N' is the [number](#) of elements in the original [matrix](#).

See Also  
-----  
[ravel](#) : Return a flattened array.  
[flat](#) : A 1-D flat iterator over the [matrix](#).

Examples  
-----  

```
>>> m = np.matrix([[1,2], [3,4]])
>>> m.flatten()
matrix([[1, 2, 3, 4]])
>>> m.flatten('F')
matrix([[1, 3, 2, 4]])
```

**getA(self)**  
Return `self` as an `ndarray` object.  
Equivalent to ``np.asarray(self)``.

Parameters  
-----  
None

Returns  
-----  
ret : [ndarray](#)  
`self` as an [ndarray](#)

Examples  
-----  

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.getA()
array([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
```

**getA1(self)**  
Return `self` as a flattened `ndarray`.  
Equivalent to ``np.asarray(x).ravel()``

```

Parameters

None

Returns

ret : ndarray
 `self`, 1-D, as an `ndarray`

Examples

>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.getA1()
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

getH(self)
 Returns the (complex) conjugate transpose of `self`.

 Equivalent to ``np.transpose(self).conjugate`` if `self` is real-valued.

Parameters

None

Returns

ret : matrix object
 complex conjugate transpose of `self`

Examples

>>> x = np.matrix(np.arange(12).reshape((3,4)))
>>> z = x - 1j*x; z
matrix([[0. +0.j, 1. -1.j, 2. -2.j, 3. -3.j],
 [4. -4.j, 5. -5.j, 6. -6.j, 7. -7.j],
 [8. -8.j, 9. -9.j, 10. -10.j, 11. -11.j]])
>>> z.getH()
matrix([[0. +0.j, 4. +4.j, 8. +8.j],
 [1. +1.j, 5. +5.j, 9. +9.j],
 [2. +2.j, 6. +6.j, 10. +10.j],
 [3. +3.j, 7. +7.j, 11. +11.j]])

getI(self)
 Returns the (multiplicative) inverse of invertible `self`.

Parameters

None

Returns

ret : matrix object
 If `self` is non-singular, `ret` is such that ``ret * self`` ==
 ``self * ret`` == ``np.matrix(np.eye(self[0,:].size))`` all return
 ``True``.

Raises

numpy.linalg.LinAlgError: Singular matrix
 If `self` is singular.

See Also

linalg.inv

Examples

>>> m = np.matrix('[[1, 2; 3, 4]); m
matrix([[1, 2],
 [3, 4]])
>>> m.getI()
matrix([[1. , 1.],
 [1.5, -0.5]])
>>> m.getI() * m
matrix([[1., 0.],
 [0., 1.]])

getT(self)
 Returns the transpose of the matrix.

 Does *not* conjugate! For the complex conjugate transpose, use ``.H``.

Parameters

None

```

```
Returns

ret : matrix object
 The (non-conjugated) transpose of the matrix.
```

See Also

```

```

[transpose](#), [getH](#)

Examples

```
----->>> m = np.matrix('[[1, 2; 3, 4]')
>>> m
matrix([[1, 2],
 [3, 4]])
>>> m.getT()
matrix([[1, 3],
 [2, 4]])
```

**max(self, axis=None, out=None)**

Return the maximum value along an axis.

Parameters

```

```

See `amax` for complete descriptions

See Also

```

```

[amax](#), [ndarray](#).max

Notes

```
-----This is the same as `ndarray.max`, but returns a `matrix` object
where `ndarray.max` would return an ndarray.
```

Examples

```
----->>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.max()
11
>>> x.max(0)
matrix([[8, 9, 10, 11]])
>>> x.max(1)
matrix([[3],
 [7],
 [11]])
```

**mean(self, axis=None, dtype=None, out=None)**

Returns the average of the [matrix](#) elements along the given axis.

Refer to `numpy.mean` for full documentation.

See Also

```

```

[numpy.mean](#)

Notes

```
-----Same as `ndarray.mean` except that, where that returns an `ndarray`,
this returns a matrix object.
```

Examples

```
----->>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.mean()
5.5
>>> x.mean(0)
matrix([[4., 5., 6., 7.]])
>>> x.mean(1)
matrix([[1.5],
 [5.5],
 [9.5]])
```

**min(self, axis=None, out=None)**

Return the minimum value along an axis.

Parameters

```

```

See `amin` for complete descriptions.

See Also  
-----  
[amin](#), [ndarray.min](#)

Notes  
-----  
This is the same as `ndarray.min`, but returns a `matrix` object where `ndarray.min` would return an [ndarray](#).

Examples  
-----  

```
>>> x = -np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, -1, -2, -3],
 [-4, -5, -6, -7],
 [-8, -9, -10, -11]])
>>> x.min()
-11
>>> x.min(0)
matrix([[-8, -9, -10, -11]])
>>> x.min(1)
matrix([[-3],
 [-7],
 [-11]])
```

**prod**(self, axis=None, dtype=None, out=None)  
Return the product of the array elements over the given axis.

Refer to `prod` for full documentation.

See Also  
-----  
[prod](#), [ndarray.prod](#)

Notes  
-----  
Same as `ndarray.prod`, except, where that returns an [ndarray](#), this returns a `matrix` object instead.

Examples  
-----  

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.prod()
0
>>> x.prod(0)
matrix([[0, 45, 120, 231]])
>>> x.prod(1)
matrix([[0],
 [840],
 [7920]])
```

**ptp**(self, axis=None, out=None)  
Peak-to-peak (maximum - minimum) value along the given axis.

Refer to `numpy.ptp` for full documentation.

See Also  
-----  
[numpy.ptp](#)

Notes  
-----  
Same as `ndarray.ptp`, except, where that would return an [ndarray](#) object, this returns a `matrix` object.

Examples  
-----  

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.ptp()
11
>>> x.ptp(0)
matrix([[8, 8, 8, 8]])
>>> x.ptp(1)
matrix([[3],
 [3],
 [3]])
```

**ravel**(self, order='C')  
Return a flattened [matrix](#).

Refer to `numpy.ravel` for more documentation.

Parameters

```

order : {'C', 'F', 'A', 'K'}, optional
 The elements of `m` are read using this index order. 'C' means to
 index the elements in C-like order, with the last axis index
 changing fastest, back to the first axis index changing slowest.
 'F' means to index the elements in Fortran-like index order, with
 the first index changing fastest, and the last index changing
 slowest. Note that the 'C' and 'F' options take no account of the
 memory layout of the underlying array, and only refer to the order
 of axis indexing. 'A' means to read the elements in Fortran-like
 index order if `m` is Fortran *contiguous* in memory, C-like order
 otherwise. 'K' means to read the elements in the order they occur
 in memory, except for reversing the data when strides are negative.
 By default, 'C' index order is used.

Returns

ret : matrix
 Return the matrix flattened to shape `(1, N)` where `N`
 is the number of elements in the original matrix.
 A copy is made only if necessary.

See Also

matrix.flatten : returns a similar output matrix but always a copy
matrix.flat : a flat iterator on the array.
numpy.ravel : related function which returns an ndarray

squeeze(self, axis=None)
 Return a possibly reshaped matrix.

 Refer to `numpy.squeeze` for more documentation.

Parameters

axis : None or int or tuple of ints, optional
 Selects a subset of the single-dimensional entries in the shape.
 If an axis is selected with shape entry greater than one,
 an error is raised.

Returns

squeezed : matrix
 The matrix, but as a (1, N) matrix if it had shape (N, 1).

See Also

numpy.squeeze : related function

Notes

If `m` has a single column then that column is returned
as the single row of a matrix. Otherwise `m` is returned.
The returned matrix is always either `m` itself or a view into `m`.
Supplying an axis keyword argument will not affect the returned matrix
but it may cause an error to be raised.

Examples

```
>>> c = np.matrix([[1], [2]])  
>>> c  
matrix([[1],  
           [2]])  
>>> c.squeeze()  
matrix([[1, 2]])  
>>> r = c.T  
>>> r  
matrix([[1, 2]])  
>>> r.squeeze()  
matrix([[1, 2]])  
>>> m = np.matrix([[1, 2], [3, 4]])  
>>> m.squeeze()  
matrix([[1, 2],  
           [3, 4]])
```


std(self, axis=None, dtype=None, out=None, ddof=0)
 Return the standard deviation of the array elements along the given axis.

 Refer to `numpy.std` for full documentation.

See Also

numpy.std

Notes

This is the same as `ndarray.std`, except that where an `ndarray` would
be returned, a matrix` object is returned instead.
```

**Examples**  
-----  
`>>> x = np.matrix(np.arange(12).reshape((3, 4)))`  
`>>> x`  
`matrix([[ 0, 1, 2, 3],`  
 `[ 4, 5, 6, 7],`  
 `[ 8, 9, 10, 11]])`  
`>>> x.std()`  
`3.4520525295346629`  
`>>> x.std(0)`  
`matrix([[ 3.26598632, 3.26598632, 3.26598632, 3.26598632]])`  
`>>> x.std(1)`  
`matrix([[ 1.11803399],`  
 `[ 1.11803399],`  
 `[ 1.11803399]])`

**sum(self, axis=None, dtype=None, out=None)**  
 Returns the sum of the `matrix` elements, along the given axis.  
 Refer to `numpy.sum` for full documentation.

**See Also**  
-----  
`numpy.sum`

**Notes**  
-----  
 This is the same as `ndarray.sum`, except that where an `ndarray` would be returned, a `matrix` object is returned instead.

**Examples**  
-----  
`>>> x = np.matrix([[1, 2], [4, 3]])`  
`>>> x.sum()`  
`10`  
`>>> x.sum(axis=1)`  
`matrix([[3],`  
 `[7]])`  
`>>> x.sum(axis=1, dtype='float')`  
`matrix([[ 3.],`  
 `[ 7.]])`  
`>>> out = np.zeros((1, 2), dtype='float')`  
`>>> x.sum(axis=1, dtype='float', out=out)`  
`matrix([[ 3.],`  
 `[ 7.]])`

**tolist(self)**  
 Return the `matrix` as a (possibly nested) list.  
 See `ndarray.tolist` for full documentation.

**See Also**  
-----  
`ndarray.tolist`

**Examples**  
-----  
`>>> x = np.matrix(np.arange(12).reshape((3,4))); x`  
`matrix([[ 0, 1, 2, 3],`  
 `[ 4, 5, 6, 7],`  
 `[ 8, 9, 10, 11]])`  
`>>> x.tolist()`  
`[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]`

**var(self, axis=None, dtype=None, out=None, ddof=0)**  
 Returns the variance of the `matrix` elements, along the given axis.  
 Refer to `numpy.var` for full documentation.

**See Also**  
-----  
`numpy.var`

**Notes**  
-----  
 This is the same as `ndarray.var`, except that where an `ndarray` would be returned, a `matrix` object is returned instead.

**Examples**  
-----  
`>>> x = np.matrix(np.arange(12).reshape((3, 4)))`  
`>>> x`  
`matrix([[ 0, 1, 2, 3],`  
 `[ 4, 5, 6, 7],`  
 `[ 8, 9, 10, 11]])`  
`>>> x.var()`  
`11.916666666666666`

```
>>> x.var(0)
matrix([[10.66666667, 10.66666667, 10.66666667, 10.66666667]])
>>> x.var(1)
matrix([[1.25],
 [1.25],
 [1.25]])
```

Static methods defined here:

[\\_\\_new\\_\\_](#)(subtype, data, dtype=None, copy=True)

Data descriptors defined here:

## A

Return `self` as an `ndarray` object.  
Equivalent to ``np.asarray(self)``.

Parameters

-----

None

Returns

-----

ret : ndarray  
`self` as an `ndarray`

Examples

-----

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.getA()
array([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
```

## A1

Return `self` as a flattened `ndarray`.  
Equivalent to ``np.asarray(x).ravel()``

Parameters

-----

None

Returns

-----

ret : ndarray  
`self`, 1-D, as an `ndarray`

Examples

-----

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]])
>>> x.getA1()
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

## H

Returns the (complex) conjugate transpose of `self`.  
Equivalent to ``np.transpose(self)` if `self` is real-valued.

Parameters

-----

None

Returns

-----

ret : matrix object  
complex conjugate transpose of `self`

Examples

-----

```
>>> x = np.matrix(np.arange(12).reshape((3,4)))
>>> z = x - 1j*x; z
matrix([[0. +0.j, 1. -1.j, 2. -2.j, 3. -3.j],
 [4. -4.j, 5. -5.j, 6. -6.j, 7. -7.j],
 [8. -8.j, 9. -9.j, 10. -10.j, 11. -11.j]])
>>> z.getH()
matrix([[0. +0.j, 4. +4.j, 8. +8.j],
 [1. +1.j, 5. +5.j, 9. +9.j],
```

```
[2. +2.j, 6. +6.j, 10.+10.j],
[3. +3.j, 7. +7.j, 11.+11.j]])
```

**I**

Returns the (multiplicative) inverse of invertible `self`.

Parameters

-----

None

Returns

-----

ret : matrix object

If `self` is non-singular, `ret` is such that ``ret \* self`` ==

``self \* ret`` == ``np.matrix(np.eye(self[0,:].size))`` all return

``True``.

Raises

-----

`numpy.linalg.LinAlgError: Singular matrix`

If `self` is singular.

See Also

-----

`linalg.inv`

Examples

-----

```
>>> m = np.matrix('1, 2; 3, 4'); m
matrix([[1, 2],
 [3, 4]])
>>> m.getI()
matrix([[-2., 1.],
 [1.5, -0.5]])
>>> m.getI() * m
matrix([[1., 0.],
 [0., 1.]])
```

**T**

Returns the transpose of the matrix.

Does \*not\* conjugate! For the complex conjugate transpose, use ``.H``.

Parameters

-----

None

Returns

-----

ret : matrix object

The (non-conjugated) transpose of the matrix.

See Also

-----

`transpose`, `getH`

Examples

-----

```
>>> m = np.matrix('1, 2; 3, 4')
>>> m
matrix([[1, 2],
 [3, 4]])
>>> m.getT()
matrix([[1, 3],
 [2, 4]])
```

**\_dict\_**

dictionary for instance variables (if defined)

Data and other attributes defined here:

---

**\_array\_priority\_** = 10.0

---

Methods inherited from [numpy.ndarray](#):

**\_abs\_(...)**

x.[\\_abs\\_](#)() <==> abs(x)

**\_add\_(...)**

x.[\\_add\\_](#)(y) <==> x+y

**\_and\_(...)**

x.[\\_and\\_](#)(y) <==> x&y

```
array(...)
 a. array(|dtype) -> reference if type unchanged, copy otherwise.

 Returns either a new reference to self if dtype is not given or a new array
 of provided data type if dtype is different from the current dtype of the
 array.

_array_prepare_(...)
 a. array_prepare(obj) -> Object of same type as ndarray object obj.

_array_wrap_(...)
 a. array wrap(obj) -> Object of same type as ndarray object a.

contains(...)
 x. contains(y) <==> y in x

copy(...)
 a. copy([order])

 Return a copy of the array.

 Parameters

 order : {'C', 'F', 'A'}, optional
 If order is 'C' (False) then the result is contiguous (default).
 If order is 'Fortran' (True) then the result has fortran order.
 If order is 'Any' (None) then the result has fortran order
 only if the array already is in fortran order.

deepcopy(...)
 a. deepcopy() -> Deep copy of array.

 Used if copy.deepcopy is called on an array.

delitem(...)
 x. delitem(y) <==> del x[y]

delslice(...)
 x. delslice(i, j) <==> del x[i:j]

 Use of negative indices is not supported.

div(...)
 x. div(y) <==> x/y

divmod(...)
 x. divmod(y) <==> divmod(x, y)

eq(...)
 x. eq(y) <==> x==y

float(...)
 x. float() <==> float(x)

floordiv(...)
 x. floordiv(y) <==> x//y

ge(...)
 x. ge(y) <==> x>=y

getslice(...)
 x. getslice(i, j) <==> x[i:j]

 Use of negative indices is not supported.

gt(...)
 x. gt(y) <==> x>y

hex(...)
 x. hex() <==> hex(x)

iadd(...)
 x. iadd(y) <==> x+=y

iand(...)
 x. iand(y) <==> x&=y
```

```
__idiv__(...)
x.__idiv__(y) <==> x/=y

__ifloordiv__(...)
x.__ifloordiv__(y) <==> x//=y

__ilshift__(...)
x.__ilshift__(y) <==> x<<=y

__imod__(...)
x.__imod__(y) <==> x%y

__index__(...)
x[y:z] <==> x[y.__index__():z.__index__()]

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__ior__(...)
x.__ior__(y) <==> x|=y

__irshift__(...)
x.__irshift__(y) <==> x>>=y

__isub__(...)
x.__isub__(y) <==> x-=y

__iter__(...)
x.__iter__() <==> iter(x)

__itruediv__(...)
x.__itruediv__(y) <==> x/=y

__ixor__(...)
x.__ixor__(y) <==> x^=y

__le__(...)
x.__le__(y) <==> x<=y

__len__(...)
x.__len__() <==> len(x)

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__lt__(...)
x.__lt__(y) <==> x<y

__mod__(...)
x.__mod__(y) <==> x%y

__ne__(...)
x.__ne__(y) <==> x!=y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x
```

```
__radd__(...)
x. __radd__(y) <==> y+x

__rand__(...)
x. __rand__(y) <==> y&x

__rdiv__(...)
x. __rdiv__(y) <==> y/x

__rdivmod__(...)
x. __rdivmod__(y) <==> divmod(y, x)

__reduce__(...)
a. __reduce__()

For pickling.

__rfloordiv__(...)
x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
x. __rlshift__(y) <==> y<<x

__rmod__(...)
x. __rmod__(y) <==> y%x

__ror__(...)
x. __ror__(y) <==> y|x

__rrshift__(...)
x. __rrshift__(y) <==> y>>x

__rshift__(...)
x. __rshift__(y) <==> x>>y

__rsub__(...)
x. __rsub__(y) <==> y-x

__rtruediv__(...)
x. __rtruediv__(y) <==> y/x

__rxor__(...)
x. __rxor__(y) <==> y^x

__setitem__(...)
x. __setitem__(i, y) <==> x[i]=y

__setslice__(...)
x. __setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

__setstate__(...)
a. __setstate__(version, shape, dtype, isfortran, rawdata)

For unpickling.

Parameters

version : int
 optional pickle version. If omitted defaults to 0.
shape : tuple
dtype : data-type
isFortran : bool
rawdata : string or list
 a binary string with the data (or a list if 'a' is an object array)

__sizeof__(...)

__sub__(...)
x. __sub__(y) <==> x-y

__truediv__(...)
x. __truediv__(y) <==> x/y

__xor__(...)
```

```
x._xor_(y) <==> x^y

argpartition(...)
a.argpartition(kth, axis=-1, kind='introselect', order=None)

 Returns the indices that would partition this array.

 Refer to `numpy.argpartition` for full documentation.

.. versionadded:: 1.8.0

See Also

numpy.argpartition : equivalent function

argsort(...)
a.argsort(axis=-1, kind='quicksort', order=None)

 Returns the indices that would sort this array.

 Refer to `numpy.argsort` for full documentation.

See Also

numpy.argsort : equivalent function

astype(...)
a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)

 Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype
 Typecode or data-type to which the array is cast.
order : {'C', 'F', 'A', 'K'}, optional
 Controls the memory layout order of the result.
 'C' means C order, 'F' means Fortran order, 'A'
 means 'F' order if all the arrays are Fortran contiguous,
 'C' order otherwise, and 'K' means as close to the
 order the array elements appear in memory as possible.
 Default is 'K'.
casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
 Controls what kind of data casting may occur. Defaults to 'unsafe'
 for backwards compatibility.

 * 'no' means the data types should not be cast at all.
 * 'equiv' means only byte-order changes are allowed.
 * 'safe' means only casts which can preserve values are allowed.
 * 'same_kind' means only safe casts or casts within a kind,
 like float64 to float32, are allowed.
 * 'unsafe' means any data conversions may be done.
subok : bool, optional
 If True, then sub-classes will be passed-through (default), otherwise
 the returned array will be forced to be a base-class array.
copy : bool, optional
 By default, astype always returns a newly allocated array. If this
 is set to false, and the `dtype`, `order`, and `subok`
 requirements are satisfied, the input array is returned instead
 of a copy.

Returns

arr_t : ndarray
 Unless `copy` is False and the other conditions for returning the input
 array are satisfied (see description for `copy` input parameter), `arr_t`
 is a new array of the same shape as the input array, with dtype, order
 given by `dtype`, `order`.

Notes

Starting in NumPy 1.9, astype method now returns an error if the string
dtype to cast to is not long enough in 'safe' casting mode to hold the max
value of integer/float array that is being casted. Previously the casting
was allowed even if the result was truncated.

Raises

ComplexWarning
 When casting from complex to float or int. To avoid this,
 one should use ``a.real.astype(t)``.

Examples

>>> x = np.array([1, 2, 2.5])
>>> x
array([1. , 2. , 2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])

byteswap(...)
a.byteswap(inplace)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by
returning a byteswapped array, optionally swapped in-place.

Parameters

inplace : bool, optional
 If ``True``, swap bytes in-place, default is ``False``.

Returns

out : ndarray
 The byteswapped array. If `inplace` is ``True``, this is
 a view to self.

Examples

>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
 dtype='|S3')

choose(...)
a.choose(choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also

numpy.choose : equivalent function

clip(...)
a.clip(min=None, max=None, out=None)

Return an array whose values are limited to ``[min, max]``.
One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See Also

numpy.clip : equivalent function

compress(...)
a.compress(condition, axis=None, out=None)

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

numpy.compress : equivalent function

conj(...)
a.conj()

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

numpy.conjugate : equivalent function

conjugate(...)
a.conjugate()
```

```
Return the complex conjugate, element-wise.
Refer to `numpy.conjugate` for full documentation.
See Also

numpy.conjugate : equivalent function
copy(...)
a.copy(order='C')
Return a copy of the array.
Parameters

order : {'C', 'F', 'A', 'K'}, optional
Controls the memory layout of the copy. 'C' means C-order,
'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
'C' otherwise. 'K' means match the layout of `a` as closely
as possible. (Note that this function and :func:`numpy.copy` are very
similar, but have different default values for their order=
arguments.)
See also

numpy.copy
numpy.copyto
Examples

>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy()
>>> x.fill(0)
>>> x
array([[0, 0, 0],
 [0, 0, 0]])
>>> y
array([[1, 2, 3],
 [4, 5, 6]])
>>> y.flags['C_CONTIGUOUS']
True
cumprod(...)
a.cumprod(axis=None, dtype=None, out=None)
Return the cumulative product of the elements along the given axis.
Refer to `numpy.cumprod` for full documentation.
See Also

numpy.cumprod : equivalent function
cumsum(...)
a.cumsum(axis=None, dtype=None, out=None)
Return the cumulative sum of the elements along the given axis.
Refer to `numpy.cumsum` for full documentation.
See Also

numpy.cumsum : equivalent function
diagonal(...)
a.diagonal(offset=0, axis1=0, axis2=1)
Return specified diagonals. In NumPy 1.9 the returned array is a
read-only view instead of a copy as in previous NumPy versions. In
a future version the read-only restriction will be removed.
Refer to :func:`numpy.diagonal` for full documentation.
See Also

numpy.diagonal : equivalent function
dot(...)
a.dot(b, out=None)
```

```
Dot product of two arrays.
Refer to `numpy.dot` for full documentation.

See Also

numpy.dot : equivalent function

Examples

>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2., 2.],
[2., 2.]])

This array method can be conveniently chained:

>>> a.dot(b).dot(b)
array([[8., 8.],
[8., 8.]])

dump(...)
a.dump(file)

Dump a pickle of the array to the specified file.
The array can be read back with pickle.load or numpy.load.

Parameters

file : str
A string naming the dump file.

dumps(...)
a.dumps()

Returns the pickle of the array as a string.
pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

fill(...)
a.fill(value)

Fill the array with a scalar value.

Parameters

value : scalar
All elements of `a` will be assigned this value.

Examples

>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])

getfield(...)
a.getfield(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in
the view are determined by the given type and the offset into the current
array in bytes. The offset needs to be such that the view dtype fits in the
array dtype; for example an array of dtype complex128 has 16-byte elements.
If taking a view with a 32-bit integer (4 bytes), the offset needs to be
between 0 and 12 bytes.

Parameters

dtype : str or dtype
The data type of the view. The dtype size of the view can not be larger
than that of the array itself.
offset : int
Number of bytes to skip before beginning the element view.

Examples

>>> x = np.diag([1.+1.j]*2)
```

```

>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j, 0.+0.j],
 [0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[1., 0.],
 [0., 2.]])
```

By choosing an offset of 8 bytes we can select the [complex](#) part of the array for our view:

```

>>> x.getfield(np.float64, offset=8)
array([[1., 0.],
 [0., 4.]])
```

**item(...)**  
a.item(\*args)

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

-----

\\*args : Arguments (variable [number](#) and type)

- \* none: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar [object](#) and returned.
- \* int\_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- \* tuple of int\_types: functions as does a [single](#) int\_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

-----

z : Standard Python scalar [object](#)  
A copy of the specified element of the array as a suitable Python scalar

**Notes**

-----

When the data type of `a` is [longdouble](#) or [clongdouble](#), [item\(\)](#) returns a scalar array [object](#) because there is no available Python scalar that would not lose information. Void arrays return a buffer [object](#) for [item\(\)](#), unless fields are defined, in which case a tuple is returned.

`item` is very similar to [a\[args\]](#), except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

**Examples**

-----

```

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

**itemset(...)**  
a.itemset(\*args)

Insert scalar into an array (scalar is cast to array's [dtype](#), if possible)

There must be at least 1 argument, and define the last argument as \*item\*. Then, ``a.itemset(\*args)`` is equivalent to but faster than ``a[args] = item``. The item should be a scalar value and `args` must select a [single](#) item in the array `a`.

**Parameters**

-----

\\*args : Arguments

- If one argument: a scalar, only used in case `a` is of size 1.
- If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a [single](#) array element location. It is either an [int](#) or a tuple.

**Notes**

-----

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an [ndarray](#), if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
 [2, 0, 3],
 [8, 5, 9]])
```

**newbyteorder(...)**

```
arr.newbyteorder(new_order='S')
```

Return the array with the same data viewed with a different [byte](#) order.

Equivalent to::

```
arr.view(arr.dtype.newbyteorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

-----

`new_order : string, optional`  
Byte order to force; a value from the [byte](#) order specifications below. `new\_order` codes can be any of:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_arr : array`  
New array [object](#) with the [dtype](#) reflecting given change to the [byte](#) order.

**nonzero(...)**

```
a.nonzero()
```

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See Also

-----

[numpy.nonzero](#) : equivalent function

**partition(...)**

```
a.partition(kth, axis=-1, kind='introselect', order=None)
```

Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

.. versionadded:: 1.8.0

Parameters

-----

`kth : int or sequence of ints`  
Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it

```
 and all equal or greater elements behind it.
 The order all elements in the partitions is undefined.
 If provided with a sequence of kth it will partition all elements
 indexed by kth of them into their sorted position at once.
axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
kind : {'introselect'}, optional
 Selection algorithm. Default is 'introselect'.
order : str or list of str, optional
 When 'a' is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.partition : Return a partitioned copy of an array.
argpartition : Indirect partition.
sort : Full sort.

Notes

See ``np.partition`` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])  
>>> a.partition(a, 3)  
>>> a  
array([[2, 1, 3, 4]])  
  
>>> a.partition((1, 3))  
array([[1, 2, 3, 4]])
```


put(...)
a.put(indices, values, mode='raise')

Set ``a.flat[n] = values[n]`` for all `n` in indices.

Refer to `numpy.put` for full documentation.

See Also

numpy.put : equivalent function

repeat(...)
a.repeat(repeats, axis=None)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also

numpy.repeat : equivalent function

reshape(...)
a.reshape(shape, order='C')

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See Also

numpy.reshape : equivalent function

resize(...)
a.resize(new_shape, refcheck=True)

Change shape and size of array in-place.

Parameters

new_shape : tuple of ints, or `n` ints
 Shape of resized array.
refcheck : bool, optional
 If False, reference count will not be checked. Default is True.

Returns

None

Raises

```

```
ValueError
 If `a` does not own its own data or references or views to it exist,
 and the data memory must be changed.

SystemError
 If the `order` keyword argument is specified. This behaviour is a
 bug in NumPy.

See Also

resize : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be
resized.

The purpose of the reference count check is to make sure you
do not use this array as a buffer for another Python object and then
reallocates the memory. However, reference counts can increase in
other ways so if you are sure that you have not shared the memory
for this array with another Python object, then you may safely set
`refcheck` to False.

Examples

Shrinking an array: array is flattened (in the order that the data are
stored in memory), resized, and reshaped:

>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
 [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
 [2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
 [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...

Unless `refcheck` is False:

>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])

round(...)
a.round(decimals=0, out=None)

Return `a` with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See Also

numpy.around : equivalent function

searchsorted(...)
a.searchsorted(v, side='left', sorter=None)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See Also

numpy.searchsorted : equivalent function
```

```
setfield(...)
a.setfield(val, dtype, offset=0)

Put a value into a specified place in a field defined by a data-type.

Place `val` into `a`'s field defined by `dtype` and beginning `offset`
bytes into the field.

Parameters

val : object
 Value to be placed in field.
dtype : dtype object
 Data-type of the field in which to place `val`.
offset : int, optional
 The number of bytes into the field at which to place `val`.

Returns

None

See Also

getfield

Examples

>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
 [3, 3, 3],
 [3, 3, 3]])
>>> x
array([[1.00000000e+000, 1.48219694e-323, 1.48219694e-323],
 [1.48219694e-323, 1.00000000e+000, 1.48219694e-323],
 [1.48219694e-323, 1.48219694e-323, 1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
```

**setflags(...)**

```
a.setflags(write=None, align=None, uic=None)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory
area used by `a` (see Notes below). The ALIGNED flag can only
be set to True if the data is actually aligned according to the type.
The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE
can only be set to True if the array owns its own memory, or the
ultimate owner of the memory exposes a writeable buffer interface,
or is a string. (The exception for string is made so that unpickling
can be done without copying memory.)
```

Parameters

```

write : bool, optional
 Describes whether or not `a` can be written to.
align : bool, optional
 Describes whether or not `a` is aligned properly for its type.
uic : bool, optional
 Describes whether or not `a` is a copy of another "base" array.
```

Notes

```

Array flags provide information about how the memory area used
for the array is to be interpreted. There are 6 Boolean flags
in use, only three of which can be changed by the user:
UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware
(as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced
by .base). When this array is deallocated, the base array will be
updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well
as the full name.
```

```

Examples

>>> y
array([[3, 1, 7],
 [2, 0, 0],
 [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True

sort(...)
a.sort(axis=-1, kind='quicksort', order=None)

Sort an array, in-place.

Parameters

axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 Sorting algorithm. Default is 'quicksort'.
order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.sort : Return a sorted copy of an array.
argsort : Indirect sort.
lexsort : Indirect stable sort on multiple keys.
searchsorted : Find elements in sorted array.
partition: Partial sort.

Notes

See ``sort`` for notes on the different sorting algorithms.

Examples

>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
 [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
 [1, 4]])

Use the `order` keyword to specify a field to use when sorting a
structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=\[\('x', 'S1'\), \('y', int\)\]\)
>>> a.sort\(order='y'\)
>>> a
array\(\[\('c', 1\), \('a', 2\)\],
 dtype=\\[\\('x', '|S1'\\), \\('y', '<i4'\\)\\]\\)\\]

swapaxes\\(...\\)
a.swapaxes\\(axis1, axis2\\)

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

numpy.swapaxes : equivalent function

```

```
take(...)
a.take(indices, axis=None, out=None, mode='raise')

 Return an array formed from the elements of `a` at the given indices.

 Refer to `numpy.take` for full documentation.

See Also

numpy.take : equivalent function

tobytes(...)
a.tobytes(order='C')

 Construct Python bytes containing the raw data bytes in the array.

 Constructs Python bytes showing a copy of the raw contents of
 data memory. The bytes object can be produced in either 'C' or 'Fortran',
 or 'Any' order (the default is 'C'-order). 'Any' order means C-order
 unless the F_CONTIGUOUS flag in the array is set, in which case it
 means 'Fortran' order.

 .. versionadded:: 1.9.0

Parameters

order : {'C', 'F', None}, optional
 Order of the data for multidimensional arrays:
 C, Fortran, or the same as for the original array.

Returns

s : bytes
 Python bytes exhibiting a copy of `a`'s raw data.

Examples

>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'

tofile(...)
a.tofile(fid, sep="", format="%s")

 Write array to a file as text or binary (default).

 Data is always written in 'C' order, independent of the order of `a`.
 The data produced by this method can be recovered using the function
 fromfile\(\).

Parameters

fid : file or str
 An open file object, or a string containing a filename.
sep : str
 Separator between array items for text output.
 If "" (empty), a binary file is written, equivalent to
 ``file.write(a.tobytes())``.
format : str
 Format string for text file output.
 Each entry in the array is formatted to text by first converting
 it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data.
Information on endianness and precision is lost, so this method is not a
good choice for files intended to archive data or transport data between
machines with different endianness. Some of these problems can be overcome
by outputting the data as text files, at the expense of speed and file
size.

toString(...)
a.toString(order='C')

 Construct Python bytes containing the raw data bytes in the array.

 Constructs Python bytes showing a copy of the raw contents of
 data memory. The bytes object can be produced in either 'C' or 'Fortran',
 or 'Any' order (the default is 'C'-order). 'Any' order means C-order
 unless the F_CONTIGUOUS flag in the array is set, in which case it
 means 'Fortran' order.
```

This function is a compatibility alias for `tobytes`. Despite its name it returns bytes not strings.

**Parameters**

-----

`order : {'C', 'F', None}, optional`  
Order of the data for multidimensional arrays:  
`C`, Fortran, or the same as for the original array.

**Returns**

-----

`s : bytes`  
Python bytes exhibiting a copy of `a`'s raw data.`

**Examples**

-----

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

**trace(...)**

`a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to `'numpy.trace'` for full documentation.

**See Also**

-----

`numpy.trace` : equivalent function

**transpose(...)**

`a.transpose(*axes)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a `matrix object`.)  
For a 2-D array, this is the usual `matrix` transpose.  
For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then  
`a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

-----

`axes : None, tuple of ints, or 'n` ints`

- \* None or no argument: reverses the order of the axes.
- \* tuple of ints: `i` in the `j`-th place in the tuple means `a`'s `i`-th axis becomes `a.transpose()`'s `j`-th axis.`
- \* `n` ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

-----

`out : ndarray`  
View of `a``, with axes suitably permuted.

**See Also**

-----

`ndarray.T` : Array property returning the array transposed.

**Examples**

-----

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
 [3, 4]])
>>> a.transpose()
array([[1, 3],
 [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
 [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
 [2, 4]])
```

**view(...)**

`a.view(dtype=None, type=None)`

New view of array with the same data.

```

Parameters

dtype : data-type or ndarray sub-class, optional
 Data-type descriptor of the returned view, e.g., float32 or int16. The
 default, None, results in the view having the same data-type as `a`.
 This argument can also be specified as an ndarray sub-class, which
 then specifies the type of the returned object (this is equivalent to
 setting the ``type`` parameter).
type : Python type, optional
 Type of the returned view, e.g., ndarray or matrix. Again, the
 default None results in type preservation.

Notes

``a.view()`` is used two different ways:

``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a view
of the array's memory with a different data-type. This can cause a
reinterpretation of the bytes of memory.

``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` just
returns an instance of `ndarray_subclass` that looks at the same array
(same shape, dtype, etc.) This does not cause a reinterpretation of the
memory.

For ``a.view(some_dtype)``, if ``some_dtype`` has a different number of
bytes per entry than the previous dtype (for example, converting a
regular array to a structured array), then the behavior of the view
cannot be predicted just from the superficial appearance of ``a`` (shown
by ``print(a)``). It also depends on exactly how ``a`` is stored in
memory. Therefore if ``a`` is C-ordered versus fortran-ordered, versus
defined as a slice or transpose, etc., the view may give different
results.

Examples

>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
Viewing array data using a different type and dtype:
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>
Creating a view on a structured array so it can be used in calculations
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
 [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
Making changes to the view changes the underlying array
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
Using a view to convert an array to a recarray:
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
Views share data:
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
Views that change the dtype size (bytes per entry) should normally be
avoided on arrays defined by slices, transposes, fortran-ordering, etc.:
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
 [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()

```

```
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([(1, 2),
 (4, 5)], dtype=[('width', '<i2'), ('length', '<i2')])
```

Data descriptors inherited from [numpy.ndarray](#):

**\_\_array\_interface\_\_**

Array protocol: Python side.

**\_\_array\_struct\_\_**

Array protocol: C-struct side.

**base**

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

**ctypes**

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

Parameters

-----

None

Returns

-----

c : Python object

Possessing attributes `data`, `shape`, `strides`, etc.

See Also

-----

[numpy.ctypeslib](#)

Notes

-----

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

\* `data`: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self.__array_interface__['data'][0]`.

\* `shape` (`c_intp*self.ndim`): A `ctypes` array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `c_int`, `c_long`, or `c_longlong` depending on the platform. The `c_intp` type is defined accordingly in `numpy.ctypeslib`. The `ctypes` array contains the shape of the underlying array.

\* `strides` (`c_intp*self.ndim`): A `ctypes` array of length `self.ndim` where the basetype is the same as for the `shape` attribute. This `ctypes` array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

\* `data_as(obj)`: Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a `ctypes` array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

\* `shape_as(obj)`: Return the shape tuple as an array of some other c-types

type. For example: `self.shape_as(ctypes.c_short)`.

\* `strides_as(obj)`: Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

Be careful using the `ctypes` attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling ```(a+b).ctypes.data_as(ctypes.c_void_p)``` returns a pointer to memory that is invalid because the array created as `(a+b)` is deallocated before the next Python statement. You can avoid this problem using either ``c=a+b`` or ``ct=(a+b).ctypes``. In the latter case, `ct` will hold a reference to the array until `ct` is deleted or re-assigned.

If the `ctypes` module is not available, then the `ctypes` attribute of array objects still returns something useful, but `ctypes` objects are not returned and errors may be raised instead. In particular, the object will still have the `as` parameter attribute which will return an integer equal to the `data` attribute.

**Examples**

```

>>> import ctypes
>>> x
array([[0, 1],
 [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

**data**

Python buffer object pointing to the start of the array's data.

**dtype**

Data-type of the array's elements.

**Parameters**

```

```

None

**Returns**

```

```

`d` : numpy `dtype` object

**See Also**

```

```

`numpy.dtype`

**Examples**

```

```

```
>>> x
array([[0, 1],
 [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

**flags**

Information about the memory layout of the array.

**Attributes**

```

```

`C_CONTIGUOUS` (C)

The data is in a single, C-style contiguous segment.

`F_CONTIGUOUS` (F)

The data is in a single, Fortran-style contiguous segment.

`OWNDATA` (O)

The array owns the memory it uses or borrows it from another object.

`WRITEABLE` (W)

The data area can be written to. Setting this to `False` locks the data, making it read-only. A view (slice, etc.) inherits `WRITEABLE` from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not

```

lock any views that already reference it, so under that circumstance it
is possible to alter the contents of a locked array via a previously
created writeable view onto it.) Attempting to change a non-writeable
array raises a RuntimeError exception.

ALIGNED (A)
 The data and all elements are aligned appropriately for the hardware.

UPDATEIFCOPY (U)
 This array is a copy of some other array. When this array is
 deallocated, the base array will be updated with the contents of
 this array.

FNC
 F_CONTIGUOUS and not C_CONTIGUOUS.

FORC
 F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)
 ALIGNED and WRITEABLE.

CARRAY (CA)
 BEHAVED and C_CONTIGUOUS.

FARRAY (FA)
 BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The `flags` object can be accessed dictionary-like (as in ``a.flags['WRITEABLE']``),
or by using lowercased attribute names (as in ``a.flags.writeable``). Short flag
names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by
the user, via direct assignment to the attribute or dictionary entry,
or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set ``False``.
- ALIGNED can only be set ``True`` if the data is truly aligned.
- WRITEABLE can only be set ``True`` if the array owns its own memory
 or the ultimate owner of the memory exposes a writeable buffer
 interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously.
This is clear for 1-dimensional arrays, but can also be true for higher
dimensional arrays.

Even for contiguous arrays a stride for a given dimension
``arr.strides[dim]`` may be *arbitrary* if ``arr.shape[dim] == 1``
or the array has no elements.
It does *not* generally hold that ``self.strides[-1] == self.itemsize``
for C-style contiguous arrays or ``self.strides[0] == self.itemsize`` for
Fortran-style contiguous arrays is true.

flat
A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not
a subclass of, Python's built-in iterator object.

See Also

flatten : Return a copy of the array collapsed into one dimension.

flatiter

Examples

>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
 [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
 [2, 5],
 [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

>>> x.flat = 3; x
array([[3, 3, 3],
 [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
 [3, 1, 3]])

```

**imag**

The imaginary part of the array.

**Examples**

```
----->>> x = np.sqrt([1+0j, 0+1j])>>> x.imagarray([0. , 0.70710678])>>> x.imag.dtypedtype('float64')
```

**itemsize**

Length of one array element in bytes.

**Examples**

```
----->>> x = np.array([1,2,3], dtype=np.float64)>>> x.itemsize8>>> x = np.array([1,2,3], dtype=np.complex128)>>> x.itemsize16
```

**nbytes**

Total bytes consumed by the elements of the array.

**Notes**

Does not include memory consumed by non-element attributes of the array object.

**Examples**

```
----->>> x = np.zeros((3,5,2), dtype=np.complex128)>>> x.nbytes480>>> np.prod(x.shape) * x.itemsize480
```

**ndim**

Number of array dimensions.

**Examples**

```
----->>> x = np.array([1, 2, 3])>>> x.ndim1>>> y = np.zeros((2, 3, 4))>>> y.ndim3
```

**real**

The real part of the array.

**Examples**

```
----->>> x = np.sqrt([1+0j, 0+1j])>>> x.realarray([1. , 0.70710678])>>> x.real.dtypedtype('float64')
```

**See Also**

`numpy.real` : equivalent function

**shape**

Tuple of array dimensions.

**Notes**

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

**Examples**

```
----->>> x = np.array([1, 2, 3, 4])>>> x.shape(4,)>>> y = np.zeros((2, 3, 4))>>> y.shape(2, 3, 4)>>> y.shape = (3, 8)>>> yarray([[0., 0., 0., 0., 0., 0., 0., 0.], [0., 0., 0., 0., 0., 0., 0., 0.],
```

```
[0., 0., 0., 0., 0., 0., 0.])
>>> y.shape = (3, 6)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged

size
Number of elements in the array.

Equivalent to ``np.prod(a.shape)``, i.e., the product of the array's
dimensions.

Examples

>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30

strides
Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ``(i[0], i[1], ..., i[n])`` in an array `a` is:::

 offset = sum(np.array(i) * a.strides)

A more detailed explanation of strides can be found in the
"ndarray.rst" file in the NumPy reference guide.

Notes

Imagine an array of 32-bit integers (each 4 bytes):::

 x = np.array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]], dtype=np.int32)

This array is stored in memory as 40 bytes, one after the other
(known as a contiguous block of memory). The strides of an array tell
us how many bytes we have to skip in memory to move to the next position
along a certain axis. For example, we have to skip 4 bytes (1 value) to
move to the next column, but 20 bytes (5 values) to get to the same
position in the next row. As such, the strides for the array `x` will be
``(20, 4)``.

See Also

numpy.lib.stride_tricks.as_strided

Examples

>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]],
 [[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

---

Data and other attributes inherited from [numpy.ndarray](#):

**\_hash\_** = None

class **memmap**([numpy.ndarray](#))

Create a memory-map to an array stored in a \*binary\* file on disk.

Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory. Numpy's `memmap`'s are array-like objects. This differs from Python's ``mmap`` module, which uses file-like objects.

This subclass of `ndarray` has some unpleasant interactions with some operations, because it doesn't quite fit properly as a subclass. An alternative to using this subclass is to create the ``mmap`` `object` yourself, then create an `ndarray` with `ndarray.__new__` directly, passing the `object` created in its `'buffer'` parameter.

This class may at some point be turned into a factory function which returns a view into an mmap buffer.

Delete the `memmap` instance to close.

#### Parameters

-----  
`filename` : `str` or file-like `object`  
    The file name or file `object` to be used as the array data buffer.  
`dtype` : data-type, optional  
    The data-type used to interpret the file contents.  
    Default is `'uint8'`.  
`mode` : {'r+', 'r', 'w+', 'c'}, optional  
    The file is opened in this mode:  
    +-----+-----+  
    | 'r' | Open existing file for reading only. |  
    +-----+-----+  
    | 'r+' | Open existing file for reading and writing. |  
    +-----+-----+  
    | 'w+' | Create or overwrite existing file for reading and writing. |  
    +-----+-----+  
    | 'c' | Copy-on-write: assignments affect data in memory, but  
    |      | changes are not saved to disk. The file on disk is  
    |      | read-only.  
    +-----+-----+

    Default is `'r+'`.

`offset` : `int`, optional  
    In the file, array data starts at this offset. Since `'offset'` is measured in bytes, it should normally be a multiple of the `byte`-size of `dtype`. When `'mode != 'r''`, even positive offsets beyond end of file are valid; The file will be extended to accommodate the additional data. By default, `memmap` will start at the beginning of the file, even if `'filename'` is a file pointer `'fp'` and `'fp.tell() != 0'`.

`shape` : tuple, optional  
    The desired shape of the array. If `'mode == 'r''` and the `number` of remaining bytes after `'offset'` is not a multiple of the `byte`-size of `dtype`, you must specify `'shape'`. By default, the returned array will be 1-D with the `number` of elements determined by file size and data-type.

`order` : {'C', 'F'}, optional  
    Specify the order of the `ndarray` memory layout:  
    :term:`row-major`, C-style or :term:`column-major`,  
    Fortran-style. This only has an effect if the shape is greater than 1-D. The default order is 'C'.

#### Attributes

-----  
`filename` : `str`  
    Path to the mapped file.  
`offset` : `int`  
    Offset position in the file.  
`mode` : `str`  
    File mode.

#### Methods

-----  
`flush`  
    Flush any changes in memory to file on disk.  
    When you delete a `memmap object`, flush is called first to write changes to disk before removing the `object`.

#### Notes

-----  
The `memmap object` can be used anywhere an `ndarray` is accepted. Given a `memmap ``fp```,  `isinstance(fp, numpy.ndarray) ``` returns `True`.

Memory-mapped arrays use the Python memory-map `object` which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

When a `memmap` causes a file to be created or extended beyond its current size in the filesystem, the contents of the new part are unspecified. On systems with POSIX filesystem semantics, the extended part will be filled with zero bytes.

Examples

```
>>> data = np.arange(12, dtype='float32')
>>> data.resize((3,4))

This example uses a temporary file so that doctest doesn't write
files to your directory. You would use a 'normal' filename.

>>> from tempfile import mkdtemp
>>> import os.path as path
>>> filename = path.join(mkdtemp(), 'newfile.dat')

Create a memmap with dtype and shape that matches our data:

>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
>>> fp
memmap([[0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.]], dtype=float32)

Write data to memmap array:

>>> fp[:] = data[:]
>>> fp
memmap([[0., 1., 2., 3.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.]], dtype=float32)

>>> fp.filename == path.abspath(filename)
True

Deletion flushes memory changes to disk before removing the object:

>>> del fp

Load the memmap and verify data was stored:

>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> newfp
memmap([[0., 1., 2., 3.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.]], dtype=float32)

Read-only memmap:

>>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> fpr.flags.writeable
False

Copy-on-write memmap:

>>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3,4))
>>> fpc.flags.writeable
True

It's possible to assign to copy-on-write array, but values are only
written into the memory copy of the array, and not written to disk:

>>> fpc
memmap([[0., 1., 2., 3.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.]], dtype=float32)
>>> fpc[0,:] = 0
>>> fpc
memmap([[0., 0., 0., 0.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.]], dtype=float32)

File on disk is unchanged:

>>> fpr
memmap([[0., 1., 2., 3.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.]], dtype=float32)

Offset into a memmap:

>>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
>>> fpo
memmap([4., 5., 6., 7., 8., 9., 10., 11.], dtype=float32)
```

Method resolution order:

[memmap](#)  
[numpy.ndarray](#)  
[\\_builtin\\_.object](#)

---

Methods defined here:

[\*\*\\_array\\_finalize\\_\(self, obj\)\*\*](#)

**flush(self)**

Write any changes in the array to the file on disk.

For further information, see '[memmap](#)'.

Parameters

-----

None

See Also

-----

[memmap](#)

---

Static methods defined here:

[\*\*\\_new\\_\(subtype, filename, dtype=<type 'numpy.uint8'>, mode='r+', offset=0, shape=None, order='C'\)\*\*](#)

---

Data descriptors defined here:

[\*\*\\_dict\\_\*\*](#)

dictionary for instance variables (if defined)

---

Data and other attributes defined here:

[\*\*\\_array\\_priority\\_\*\* = -100.0](#)

---

Methods inherited from [numpy.ndarray](#):

[\*\*\\_abs\\_\(...\)\*\*](#)

x. [\\_abs\\_\(\)](#) <==> abs(x)

[\*\*\\_add\\_\(...\)\*\*](#)

x. [\\_add\\_\(y\)](#) <==> x+y

[\*\*\\_and\\_\(...\)\*\*](#)

x. [\\_and\\_\(y\)](#) <==> x&y

[\*\*\\_array\\_\(...\)\*\*](#)

a. [\\_array\\_\(|dtype\)](#) -> reference if type unchanged, copy otherwise.

Returns either a new reference to self if [dtype](#) is not given or a new array of provided data type if [dtype](#) is different from the current [dtype](#) of the array.

[\*\*\\_array\\_prepare\\_\(...\)\*\*](#)

a. [\\_array\\_prepare\\_\(obj\)](#) -> Object of same type as [ndarray object](#) obj.

[\*\*\\_array\\_wrap\\_\(...\)\*\*](#)

a. [\\_array\\_wrap\\_\(obj\)](#) -> Object of same type as [ndarray object](#) a.

[\*\*\\_contains\\_\(...\)\*\*](#)

x. [\\_contains\\_\(y\)](#) <==> y in x

[\*\*\\_copy\\_\(...\)\*\*](#)

a. [\\_copy\\_\(\[order\]\)](#)

Return a copy of the array.

Parameters

-----

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

If order is 'C' (False) then the result is contiguous (default).

If order is 'Fortran' (True) then the result has fortran order.

If order is 'Any' (None) then the result has fortran order

only if the array already is in fortran order.

**deepcopy**(...)  
a. deepcopy() -> Deep copy of array.

Used if copy.deepcopy is called on an array.

**delitem**(...)  
x. delitem(y) <==> del x[y]

**delslice**(...)  
x. delslice(i, j) <==> del x[i:j]

Use of negative indices is not supported.

**div**(...)  
x. div(y) <==> x/y

**divmod**(...)  
x. divmod(y) <==> divmod(x, y)

**eq**(...)  
x. eq(y) <==> x==y

**float**(...)  
x. float() <==> float(x)

**floordiv**(...)  
x. floordiv(y) <==> x//y

**ge**(...)  
x. ge(y) <==> x>=y

**getitem**(...)  
x. getitem(y) <==> x[y]

**getslice**(...)  
x. getslice(i, j) <==> x[i:j]

Use of negative indices is not supported.

**gt**(...)  
x. gt(y) <==> x>y

**hex**(...)  
x. hex() <==> hex(x)

**iadd**(...)  
x. iadd(y) <==> x+=y

**iand**(...)  
x. iand(y) <==> x&=y

**idiv**(...)  
x. idiv(y) <==> x/=y

**ifloordiv**(...)  
x. ifloordiv(y) <==> x//=y

**ilshift**(...)  
x. ilshift(y) <==> x<<=y

**imod**(...)  
x. imod(y) <==> x%&=y

**imul**(...)  
x. imul(y) <==> x\*&=y

**index**(...)  
x[y:z] <==> x[y. index() : z. index()]

**int**(...)  
x. int() <==> int(x)

**invert**(...)  
x. invert() <==> ~x

```
ior(...)
x.ior(y) <==> x|=y

ipow(...)
x.ipow(y) <==> x**=y

irshift(...)
x.irshift(y) <==> x>>=y

isub(...)
x.isub(y) <==> x-=y

iter(...)
x.iter() <==> iter(x)

itruediv(...)
x.itruediv(y) <==> x/=y

ixor(...)
x.ixor(y) <==> x^=y

le(...)
x.le(y) <==> x<=y

len(...)
x.len() <==> len(x)

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

lt(...)
x.lt(y) <==> x<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

ne(...)
x.ne(y) <==> x!=y

neg(...)
x.neg() <==> -x

nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)
```

```
__reduce__(...)
 a. __reduce__()

 For pickling.

__repr__(...)
 x. __repr__() <==> repr(x)

__rfloordiv__(...)
 x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
 x. __rlshift__(y) <==> y<<x

__rmod__(...)
 x. __rmod__(y) <==> y%x

__rmul__(...)
 x. __rmul__(y) <==> y*x

__ror__(...)
 x. __ror__(y) <==> y|x

__rpow__(...)
 y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x. __rrshift__(y) <==> y>>x

__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setitem__(...)
 x. __setitem__(i, y) <==> x[i]=y

__setslice__(...)
 x. __setslice__(i, j, y) <==> x[i:j]=y

 Use of negative indices is not supported.

__setstate__(...)
 a. __setstate__(version, shape, dtype, isfortran, rawdata)

 For unpickling.

 Parameters

 version : int
 optional pickle version. If omitted defaults to 0.
 shape : tuple
 dtype : data-type
 isFortran : bool
 rawdata : string or list
 a binary string with the data (or a list if 'a' is an object array)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
```

```
x.xor(y) <==> x^y

all(...)
a.all(axis=None, out=None, keepdims=False)
 Returns True if all elements evaluate to True.
 Refer to `numpy.all` for full documentation.
 See Also

 numpy.all : equivalent function

any(...)
a.any(axis=None, out=None, keepdims=False)
 Returns True if any of the elements of `a` evaluate to True.
 Refer to `numpy.any` for full documentation.
 See Also

 numpy.any : equivalent function

argmax(...)
a.argmax(axis=None, out=None)
 Return indices of the maximum values along the given axis.
 Refer to `numpy.argmax` for full documentation.
 See Also

 numpy.argmax : equivalent function

argmin(...)
a.argmin(axis=None, out=None)
 Return indices of the minimum values along the given axis of `a`.
 Refer to `numpy.argmin` for detailed documentation.
 See Also

 numpy.argmin : equivalent function

argpartition(...)
a.argpartition(kth, axis=-1, kind='introselect', order=None)
 Returns the indices that would partition this array.
 Refer to `numpy.argpartition` for full documentation.
 .. versionadded:: 1.8.0
 See Also

 numpy.argpartition : equivalent function

argsort(...)
a.argsort(axis=-1, kind='quicksort', order=None)
 Returns the indices that would sort this array.
 Refer to `numpy.argsort` for full documentation.
 See Also

 numpy.argsort : equivalent function

astype(...)
a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)
 Copy of the array, cast to a specified type.
 Parameters

 dtype : str or dtype
 Typecode or data-type to which the array is cast.
 order : {'C', 'F', 'A', 'K'}, optional
 Controls the memory layout order of the result.
 'C' means C order, 'F' means Fortran order, 'A'
 means 'F' order if all the arrays are Fortran contiguous,
 'C' order otherwise, and 'K' means as close to the
 order the array elements appear in memory as possible.
```

```
 Default is 'K'.
casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
 Controls what kind of data casting may occur. Defaults to 'unsafe'
 for backwards compatibility.

 * 'no' means the data types should not be cast at all.
 * 'equiv' means only byte-order changes are allowed.
 * 'safe' means only casts which can preserve values are allowed.
 * 'same_kind' means only safe casts or casts within a kind,
 like float64 to float32, are allowed.
 * 'unsafe' means any data conversions may be done.

subok : bool, optional
 If True, then sub-classes will be passed-through (default), otherwise
 the returned array will be forced to be a base-class array.

copy : bool, optional
 By default, astype always returns a newly allocated array. If this
 is set to false, and the dtype, order, and subok
 requirements are satisfied, the input array is returned instead
 of a copy.

Returns

arr_t : ndarray
 Unless `copy` is False and the other conditions for returning the input
 array are satisfied (see description for `copy` input parameter), `arr_t`
 is a new array of the same shape as the input array, with dtype, order
 given by dtype, order.
```

Notes

-----

Starting in NumPy 1.9, astype method now returns an error if the string [dtype](#) to cast to is not long enough in 'safe' casting mode to hold the max value of [integer](#)/[float](#) array that is being casted. Previously the casting was allowed even if the result was truncated.

Raises

-----

[ComplexWarning](#)

When casting from [complex](#) to [float](#) or [int](#). To avoid this,
one should use ``a.real.astype(t)``.

Examples

-----

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

**byteswap(...)**

a.[byteswap](#)(inplace)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by
returning a byteswapped array, optionally swapped in-place.

Parameters

-----

inplace : bool, optional
 If ``True``, swap bytes in-place, default is ``False``.

Returns

-----

out : [ndarray](#)
 The byteswapped array. If `inplace` is ``True``, this is
 a view to self.

Examples

-----

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
 dtype='|S3')
```

**choose(...)**

a.[choose](#)(choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.  
Refer to `numpy.choose` for full documentation.

See Also  
-----  
[numpy.choose](#) : equivalent function

**clip(...)**  
[a.clip](#)(min=None, max=None, out=None)

Return an array whose values are limited to ``[min, max]``.  
One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See Also  
-----  
[numpy.clip](#) : equivalent function

**compress(...)**  
[a.compress](#)(condition, axis=None, out=None)

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also  
-----  
[numpy.compress](#) : equivalent function

**conj(...)**  
[a.conj](#)()

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also  
-----  
[numpy.conjugate](#) : equivalent function

**conjugate(...)**  
[a.conjugate](#)()

Return the [complex](#) conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See Also  
-----  
[numpy.conjugate](#) : equivalent function

**copy(...)**  
[a.copy](#)(order='C')

Return a copy of the array.

Parameters  
-----  
`order : {'C', 'F', 'A', 'K'}`, optional  
Controls the memory layout of the copy. 'C' means C-order,  
'F' means F-order, 'A' means 'F' if 'a' is Fortran contiguous,  
'C' otherwise. 'K' means match the layout of 'a' as closely  
as possible. (Note that this function and :func:`numpy.copy` are very  
similar, but have different default values for their order=  
arguments.)

See also  
-----  
[numpy.copy](#)  
[numpy.copyto](#)

Examples  
-----

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy()
>>> x.fill(0)
>>> x
array([[0, 0, 0],
 [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
 [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True

cumprod(...)
a.cumprod(axis=None, dtype=None, out=None)

 Return the cumulative product of the elements along the given axis.

 Refer to `numpy.cumprod` for full documentation.

 See Also

 numpy.cumprod : equivalent function

cumsum(...)
a.cumsum(axis=None, dtype=None, out=None)

 Return the cumulative sum of the elements along the given axis.

 Refer to `numpy.cumsum` for full documentation.

 See Also

 numpy.cumsum : equivalent function

diagonal(...)
a.diagonal(offset=0, axis1=0, axis2=1)

 Return specified diagonals. In NumPy 1.9 the returned array is a
 read-only view instead of a copy as in previous NumPy versions. In
 a future version the read-only restriction will be removed.

 Refer to :func:`numpy.diagonal` for full documentation.

 See Also

 numpy.diagonal : equivalent function

dot(...)
a.dot(b, out=None)

 Dot product of two arrays.

 Refer to `numpy.dot` for full documentation.

 See Also

 numpy.dot : equivalent function

 Examples

 >>> a = np.eye(2)
 >>> b = np.ones((2, 2)) * 2
 >>> a.dot(b)
 array([[2., 2.],
 [2., 2.]])

 This array method can be conveniently chained:

 >>> a.dot(b).dot(b)
 array([[8., 8.],
 [8., 8.]])
```

**dump(...)**

a.dump(file)

Dump a pickle of the array to the specified file.  
The array can be read back with pickle.load or numpy.load.

Parameters

-----

file : str  
A string naming the dump file.

**dumps(...)**

a.dumps()

Returns the pickle of the array as a string.  
pickle.loads or numpy.loads will convert the string back to an array.

Parameters

-----

```
None

fill(...)
a.fill(value)

 Fill the array with a scalar value.

Parameters

value : scalar
 All elements of `a` will be assigned this value.

Examples

>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])

flatten(...)
a.flatten(order='C')

 Return a copy of the array collapsed into one dimension.

Parameters

order : {'C', 'F', 'A', 'K'}, optional
 'C' means to flatten in row-major (C-style) order.
 'F' means to flatten in column-major (Fortran-style) order.
 'A' means to flatten in column-major order if `a` is Fortran *contiguous* in memory,
 row-major order otherwise. 'K' means to flatten `a` in the order the elements occur in memory.
 The default is 'C'.

Returns

y : ndarray
 A copy of the input array, flattened to one dimension.

See Also

ravel : Return a flattened array.
flat : A 1-D flat iterator over the array.

Examples

>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])

getfield(...)
a.getfield(dtype, offset=0)

 Returns a field of the given array as a certain type.

 A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype
 The data type of the view. The dtype size of the view can not be larger than that of the array itself.
offset : int
 Number of bytes to skip before beginning the element view.

Examples

>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j, 0.+0.j],
 [0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[1., 0.],
 [0., 2.]])
```

By choosing an offset of 8 bytes we can select the [complex](#) part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1., 0.],
 [0., 4.]])
```

**item(...)**  
`a.item(*args)`

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

-----

\*args : Arguments (variable [number](#) and type)

- \* none: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar [object](#) and returned.
- \* int\_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- \* tuple of int\_types: functions as does a [single](#) int\_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

-----

z : Standard Python scalar [object](#)  
A copy of the specified element of the array as a suitable Python scalar

**Notes**

-----

When the data type of `a` is [longdouble](#) or [clongdouble](#), [item\(\)](#) returns a scalar array [object](#) because there is no available Python scalar that would not lose information. Void arrays return a buffer [object](#) for [item\(\)](#), unless fields are defined, in which case a tuple is returned.

`item` is very similar to `a[args]`, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

**Examples**

-----

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
```

```
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

**itemset(...)**  
`a.itemset(*args)`

Insert scalar into an array (scalar is cast to array's [dtype](#), if possible)

There must be at least 1 argument, and define the last argument as \*item\*. Then, ``a.[itemset](#)(\*args)`` is equivalent to but faster than ``a[args] = item``. The item should be a scalar value and `args` must select a [single](#) item in the array `a`.

**Parameters**

-----

\*args : Arguments

- If one argument: a scalar, only used in case `a` is of size 1.
- If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a [single](#) array element location. It is either an [int](#) or a tuple.

**Notes**

-----

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

```
Examples

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
 [2, 0, 3],
 [8, 5, 9]])

max(...)
a.max(axis=None, out=None)

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See Also

numpy.amax : equivalent function

mean(...)
a.mean(axis=None, dtype=None, out=None, keepdims=False)

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also

numpy.mean : equivalent function

min(...)
a.min(axis=None, out=None, keepdims=False)

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also

numpy.amin : equivalent function

newbyteorder(...)
arr.newbyteorder(new_order='S')

Return the array with the same data viewed with a different byte order.

Equivalent to::
 arr.view(arr.dtype.newbyteorder(new_order))

Changes are also made in all fields and sub-arrays of the array data
type.

Parameters

new_order : string, optional
 Byte order to force; a value from the byte order specifications
 below. `new_order` codes can be any of:
 * 'S' - swap dtype from current to opposite endian
 * {'<', '<'} - little endian
 * {'>', '>'} - big endian
 * {'=', '='} - native order
 * {'|', '|'} - ignore (no change to byte order)

The default value ('S') results in swapping the current
byte order. The code does a case-insensitive check on the first
letter of `new_order` for the alternatives above. For example,
any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_arr : array
 New array object with the dtype reflecting given change to the
 byte order.
```

```
nonzero(...)
a.nonzero\(\)

 Return the indices of the elements that are non-zero.

 Refer to `numpy.nonzero` for full documentation.

See Also

numpy.nonzero : equivalent function

partition(...)
a.partition(kth, axis=-1, kind='introselect', order=None)

 Rearranges the elements in the array in such a way that value of the
 element in kth position is in the position it would be in a sorted array.
 All elements smaller than the kth element are moved before this element and
 all equal or greater are moved behind it. The ordering of the elements in
 the two partitions is undefined.

 .. versionadded:: 1.8.0

Parameters

kth : int or sequence of ints
 Element index to partition by. The kth element value will be in its
 final sorted position and all smaller elements will be moved before it
 and all equal or greater elements behind it.
 The order all elements in the partitions is undefined.
 If provided with a sequence of kth it will partition all elements
 indexed by kth of them into their sorted position at once.
axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
kind : {'introselect'}, optional
 Selection algorithm. Default is 'introselect'.
order : str or list of str, optional
 When 'a' is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.partition : Return a partitioned copy of an array.
argpartition : Indirect partition.
sort : Full sort.

Notes

See ``np.partition`` for notes on the different algorithms.

Examples

>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])

prod(...)
a.prod(axis=None, dtype=None, out=None, keepdims=False)

 Return the product of the array elements over the given axis

 Refer to `numpy.prod` for full documentation.

See Also

numpy.prod : equivalent function

ptp(...)
a.ptp(axis=None, out=None)

 Peak to peak (maximum - minimum) value along a given axis.

 Refer to `numpy.ptp` for full documentation.

See Also

numpy.ptp : equivalent function

put(...)
a.put(indices, values, mode='raise')
```

```
Set ``a.flat[n] = values[n]`` for all `n` in indices.
Refer to `numpy.put` for full documentation.
See Also

numpy.put : equivalent function
ravel(...)
a.ravel([order])
Return a flattened array.
Refer to `numpy.ravel` for full documentation.
See Also

numpy.ravel : equivalent function
ndarray.flat : a flat iterator on the array.
repeat(...)
a.repeat(repeats, axis=None)
Repeat elements of an array.
Refer to `numpy.repeat` for full documentation.
See Also

numpy.repeat : equivalent function
reshape(...)
a.reshape(shape, order='C')
Returns an array containing the same data with a new shape.
Refer to `numpy.reshape` for full documentation.
See Also

numpy.reshape : equivalent function
resize(...)
a.resize(new_shape, refcheck=True)
Change shape and size of array in-place.
Parameters

new_shape : tuple of ints, or `n` ints
 Shape of resized array.
refcheck : bool, optional
 If False, reference count will not be checked. Default is True.
Returns

None
Raises

ValueError
 If `a` does not own its own data or references or views to it exist,
 and the data memory must be changed.
SystemError
 If the `order` keyword argument is specified. This behaviour is a
 bug in NumPy.
See Also

resize : Return a new array with the specified shape.
Notes

This reallocates space for the data area if necessary.
Only contiguous arrays (data elements consecutive in memory) can be
resized.
The purpose of the reference count check is to make sure you
do not use this array as a buffer for another Python object and then
reallocates the memory. However, reference counts can increase in
other ways so if you are sure that you have not shared the memory
for this array with another Python object, then you may safely set
'refcheck' to False.
```

```
Examples

Shrinking an array: array is flattened (in the order that the data are
stored in memory), resized, and reshaped:

>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[[0],
 [1]]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[[0],
 [2]]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
 [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...

Unless `refcheck` is False:

>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])

round(...)
a.round(decimals=0, out=None)

 Return `a` with each element rounded to the given number of decimals.

 Refer to `numpy.around` for full documentation.

 See Also

 numpy.around : equivalent function

searchsorted(...)
a.searchsorted(v, side='left', sorter=None)

 Find indices where elements of v should be inserted in a to maintain order.

 For full documentation, see `numpy.searchsorted`.

 See Also

 numpy.searchsorted : equivalent function

setfield(...)
a.setfield(val, dtype, offset=0)

 Put a value into a specified place in a field defined by a data-type.

 Place `val` into `a`'s field defined by `dtype` and beginning `offset`
 bytes into the field.

 Parameters

 val : object
 Value to be placed in field.
 dtype : dtype object
 Data-type of the field in which to place `val`.
 offset : int, optional
 The number of bytes into the field at which to place `val`.

 Returns

 None

 See Also

 getfield
```

```

Examples

>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
 [3, 3, 3],
 [3, 3, 3]])
>>> x
array([[1.0000000e+000, 1.48219694e-323, 1.48219694e-323],
 [1.48219694e-323, 1.0000000e+000, 1.48219694e-323],
 [1.48219694e-323, 1.48219694e-323, 1.0000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
```

**setflags(...)**

```
a.setflags(write=None, align=None, uic=None)
```

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by `a` (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

```

```

- write : bool, optional
 Describes whether or not `a` can be written to.
- align : bool, optional
 Describes whether or not `a` is aligned properly for its type.
- uic : bool, optional
 Describes whether or not `a` is a copy of another "base" array.

Notes

-----

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```

```

```

>>> y
array([[3, 1, 7],
 [2, 0, 0],
 [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
ValueError: cannot set UPDATEIFCOPY flag to True

sort(...)
 a.sort(axis=-1, kind='quicksort', order=None)

 Sort an array, in-place.

 Parameters

 axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
 kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 Sorting algorithm. Default is 'quicksort'.
 order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

 See Also

 numpy.sort : Return a sorted copy of an array.
 argsort : Indirect sort.
 lexsort : Indirect stable sort on multiple keys.
 searchsorted : Find elements in sorted array.
 partition: Partial sort.

 Notes

 See ``sort`` for notes on the different sorting algorithms.

 Examples

 >>> a = np.array([[1,4], [3,1]])
 >>> a.sort(axis=1)
 >>> a
 array([[1, 4],
 [1, 3]])
 >>> a.sort(axis=0)
 >>> a
 array([[1, 3],
 [1, 4]])

 Use the `order` keyword to specify a field to use when sorting a
 structured array:

 >>> a = np.array([('a', 2), ('c', 1)], dtype[('x', 'S1'), ('y', int)])
 >>> a.sort(order='y')
 >>> a
 array([('c', 1), ('a', 2)],
 dtype[('x', '|S1'), ('y', '<i4')])

squeeze(...)
 a.squeeze(axis=None)

 Remove single-dimensional entries from the shape of `a`.

 Refer to `numpy.squeeze` for full documentation.

 See Also

 numpy.squeeze : equivalent function

std(...)
 a.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

 Returns the standard deviation of the array elements along given axis.

 Refer to `numpy.std` for full documentation.

 See Also

 numpy.std : equivalent function

sum(...)
 a.sum(axis=None, dtype=None, out=None, keepdims=False)

 Return the sum of the array elements over the given axis.

 Refer to `numpy.sum` for full documentation.

 See Also

 numpy.sum : equivalent function
```

```
swapaxes(...)
a.swapaxes(axis1, axis2)

 Return a view of the array with `axis1` and `axis2` interchanged.

 Refer to `numpy.swapaxes` for full documentation.

 See Also

 numpy.swapaxes : equivalent function

take(...)
a.take(indices, axis=None, out=None, mode='raise')

 Return an array formed from the elements of `a` at the given indices.

 Refer to `numpy.take` for full documentation.

 See Also

 numpy.take : equivalent function

tobytes(...)
a.tobytes(order='C')

 Construct Python bytes containing the raw data bytes in the array.

 Constructs Python bytes showing a copy of the raw contents of
 data memory. The bytes object can be produced in either 'C' or 'Fortran',
 or 'Any' order (the default is 'C'-order). 'Any' order means C-order
 unless the F_CONTIGUOUS flag in the array is set, in which case it
 means 'Fortran' order.

 .. versionadded:: 1.9.0

 Parameters

 order : {'C', 'F', None}, optional
 Order of the data for multidimensional arrays:
 C, Fortran, or the same as for the original array.

 Returns

 s : bytes
 Python bytes exhibiting a copy of `a`'s raw data.

 Examples

 >>> x = np.array([[0, 1], [2, 3]])
 >>> x.tobytes()
 b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
 >>> x.tobytes('C') == x.tobytes()
 True
 >>> x.tobytes('F')
 b'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'

tofile(...)
a.tofile(fid, sep="", format="%s")

 Write array to a file as text or binary (default).

 Data is always written in 'C' order, independent of the order of `a`.
 The data produced by this method can be recovered using the function
 fromfile().

 Parameters

 fid : file or str
 An open file object, or a string containing a filename.
 sep : str
 Separator between array items for text output.
 If "" (empty), a binary file is written, equivalent to
 ``file.write(a.tobytes())``.
 format : str
 Format string for text file output.
 Each entry in the array is formatted to text by first converting
 it to the closest Python type, and then using "format" % item.

 Notes

 This is a convenience function for quick storage of array data.
 Information on endianness and precision is lost, so this method is not a
 good choice for files intended to archive data or transport data between
 machines with different endianness. Some of these problems can be overcome
 by outputting the data as text files, at the expense of speed and file
 size.
```

```
tolist(...)
a.tolist\(\)

 Return the array as a (possibly nested) list.

 Return a copy of the array data as a (nested) Python list.
 Data items are converted to the nearest compatible Python type.

Parameters

none

Returns

y : list
 The possibly nested list of array elements.

Notes

The array may be recreated, ``a = np.array(a.tolist\(\))``.

Examples

>>> a = np.array([1, 2])
>>> a.tolist\(\)
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist\(\)
[[1, 2], [3, 4]]
```

  

```
tostring(...)
a.tostring\(order='C'\)

 Construct Python bytes containing the raw data bytes in the array.

 Constructs Python bytes showing a copy of the raw contents of
 data memory. The bytes object can be produced in either 'C' or 'Fortran',
 or 'Any' order (the default is 'C'-order). 'Any' order means C-order
 unless the F_CONTIGUOUS flag in the array is set, in which case it
 means 'Fortran' order.

 This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

Parameters

order : {'C', 'F', None}, optional
 Order of the data for multidimensional arrays:
 C, Fortran, or the same as for the original array.

Returns

s : bytes
 Python bytes exhibiting a copy of `a`'s raw data.

Examples

>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes\(\)
b'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes\('C'\) == x.tobytes\(\)
True
>>> x.tobytes\('F'\)
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x00\x03\x00\x00\x00'
```

  

```
trace(...)
a.trace\(offset=0, axis1=0, axis2=1, dtype=None, out=None\)

 Return the sum along diagonals of the array.

 Refer to `numpy.trace` for full documentation.

See Also

numpy.trace : equivalent function
```

  

```
transpose(...)
a.transpose\(*axes\)

 Returns a view of the array with axes transposed.

 For a 1-D array, this has no effect. (To change between column and
 row vectors, first cast the 1-D array into a matrix object.)
 For a 2-D array, this is the usual matrix transpose.
 For an n-D array, if axes are given, their order indicates how the
 axes are permuted (see Examples). If axes are not provided and
```

```

``a.shape = (i[0], i[1], ... i[n-2], i[n-1])``, then
``a.transpose\(\).shape = (i[n-1], i[n-2], ... i[1], i[0])``.

Parameters

axes : None, tuple of ints, or `n` ints

* None or no argument: reverses the order of the axes.

* tuple of ints: `i` in the `j`-th place in the tuple means `a`'s
 `i`-th axis becomes `a.transpose\(\)`'s `j`-th axis.

* `n` ints: same as an n-tuple of the same ints (this form is
 intended simply as a "convenience" alternative to the tuple form)

Returns

out : ndarray
 View of `a`, with axes suitably permuted.

See Also

ndarray.T : Array property returning the array transposed.

Examples

>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
 [3, 4]])
>>> a.transpose\(\)
array([[1, 3],
 [2, 4]])
>>> a.transpose\(\(1, 0\)\)
array([[1, 3],
 [2, 4]])
>>> a.transpose\(1, 0\)
array([[1, 3],
 [2, 4]])

var(...)
a.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also

numpy.var : equivalent function

view(...)
a.view(dtype=None, type=None)

New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional
 Data-type descriptor of the returned view, e.g., float32 or int16. The
 default, None, results in the view having the same data-type as `a`.
 This argument can also be specified as an ndarray sub-class, which
 then specifies the type of the returned object (this is equivalent to
 setting the ``type`` parameter).
type : Python type, optional
 Type of the returned view, e.g., ndarray or matrix. Again, the
 default None results in type preservation.

Notes

``a.view\(\)`` is used two different ways:

``a.view\(some_dtype\)`` or ``a.view\(dtype=some_dtype\)`` constructs a view
of the array's memory with a different data-type. This can cause a
reinterpretation of the bytes of memory.

``a.view\(ndarray_subclass\)`` or ``a.view\(type=ndarray_subclass\)`` just
returns an instance of `ndarray_subclass` that looks at the same array
(same shape, dtype, etc.) This does not cause a reinterpretation of the
memory.

For ``a.view\(some_dtype\)``, if ``some_dtype`` has a different number of
bytes per entry than the previous dtype (for example, converting a
regular array to a structured array), then the behavior of the view
cannot be predicted just from the superficial appearance of ``a`` (shown
by ``print(a)``). It also depends on exactly how ``a`` is stored in
memory. Therefore if ``a`` is C-ordered versus fortran-ordered, versus
defined as a slice or transpose, etc., the view may give different

```

```

results.

Examples

>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
Viewing array data using a different type and dtype:
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>
Creating a view on a structured array so it can be used in calculations
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
 [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])

Making changes to the view changes the underlying array
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]

Using a view to convert an array to a recarray:
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)

Views share data:
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)

Views that change the dtype size (bytes per entry) should normally be
avoided on arrays defined by slices, transposes, fortran-ordering, etc.:
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
 [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([(1, 2)],
 [(4, 5)], dtype=[('width', '<i2'), ('length', '<i2')])
```

Data descriptors inherited from [numpy.ndarray](#):

**T**  
Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

```

Examples

>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[1., 2.],
 [3., 4.]])
>>> x.T
array([[1., 3.],
 [2., 4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([1., 2., 3., 4.])
>>> x.T
array([1., 2., 3., 4.])
```

**\_array\_interface\_**  
Array protocol: Python side.

**\_array\_struct\_**  
Array protocol: C-struct side.

**base**

Base object if memory is from some other object.

**Examples**

-----

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

-----

None

**Returns**

-----

c : Python object  
Possessing attributes data, shape, strides, etc.

**See Also**

-----

`numpy.ctypeslib`

**Notes**

-----

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

\* data: A pointer to the memory area of the array as a Python integer.  
This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

\* shape (c\_intp\*`self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `c_int`, `c_long`, or `c_longlong` depending on the platform. The `c_intp` type is defined accordingly in `numpy.ctypeslib`. The ctypes array contains the shape of the underlying array.

\* strides (c\_intp\*`self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

\* data\_as(obj): Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data:  
`self.data_as(ctypes.POINTER(ctypes.c_double))`.

\* shape\_as(obj): Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

\* strides\_as(obj): Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling ```(a+b).ctypes.data_as(ctypes.c_void_p)``` returns a pointer to memory that is invalid because the array created as `(a+b)` is deallocated before the next Python statement. You can avoid this problem using either ```c=a+b``` or ```ct=(a+b).ctypes```. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects

are not returned and errors may be raised instead. In particular, the object will still have the `as` parameter attribute which will return an integer equal to the `data` attribute.

#### Examples

```

>>> import ctypes
>>> x
array([[0, 1,
 [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

### **data**

Python buffer object pointing to the start of the array's data.

### **dtype**

Data-type of the array's elements.

#### Parameters

-----

None

#### Returns

-----

d : numpy dtype object

#### See Also

-----

`numpy.dtype`

#### Examples

```

>>> x
array([[0, 1,
 [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

### **flags**

Information about the memory layout of the array.

#### Attributes

-----

##### C\_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

##### F\_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

##### OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

##### WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a `RuntimeError` exception.

##### ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

##### UPDATEIFCOPY (U)

This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

##### FNC

F\_CONTIGUOUS and not C\_CONTIGUOUS.

##### FORC

F\_CONTIGUOUS or C\_CONTIGUOUS (one-segment test).

##### BEHAVED (B)

ALIGNED and WRITEABLE.  
CARRAY (CA)  
BEHAVED and C\_CONTIGUOUS.  
FARRAY (FA)  
BEHAVED and F\_CONTIGUOUS and not C\_CONTIGUOUS.

#### Notes

The `flags` object can be accessed dictionary-like (as in ``a.flags['WRITEABLE']``), or by using lowercased attribute names (as in ``a.flags.writeable``). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set ``False``.
- ALIGNED can only be set ``True`` if the data is truly aligned.
- WRITEABLE can only be set ``True`` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension ``arr.strides[dim]`` may be \*arbitrary\* if ``arr.shape[dim] == 1`` or the array has no elements. It does \*not\* generally hold that ``self.strides[-1] == self.itemsize`` for C-style contiguous arrays or ``self.strides[0] == self.itemsize`` for Fortran-style contiguous arrays is true.

## flat

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

#### See Also

flatten : Return a copy of the array collapsed into one dimension.

## flatiter

#### Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
 [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
 [2, 5],
 [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
 [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
 [3, 1, 3]])
```

## imag

The imaginary part of the array.

#### Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

## itemsize

Length of one array element in bytes.

#### Examples

```

>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

**nbytes**

Total bytes consumed by the elements of the array.

**Notes**

Does not include memory consumed by non-element attributes of the array object.

**Examples**

```

>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**ndim**

Number of array dimensions.

**Examples**

```

>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

**real**

The real part of the array.

**Examples**

```

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

**See Also**

`numpy.real` : equivalent function

**shape**

Tuple of array dimensions.

**Notes**

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

**Examples**

```

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

**size**

Number of elements in the array.

Equivalent to ```np.prod(a.shape)```, i.e., the product of the array's dimensions.

**Examples**

```

>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
```

```

>>> x.size
30
>>> np.prod(x.shape)
30

strides
 Tuple of bytes to step in each dimension when traversing an array.

 The byte offset of element ``(i[0], i[1], ..., i[n])`` in an array `a` is::

 offset = sum(np.array(i) * a.strides)

 A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

Notes

Imagine an array of 32-bit integers (each 4 bytes)::

 x = np.array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]], dtype=np.int32)

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be ``(20, 4)``.

See Also

numpy.lib.stride_tricks.as_strided

Examples

>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]],
 [[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813

```

Data and other attributes inherited from [numpy.ndarray](#):

**\_hash\_** = None

```

class ndarray(__builtin__.object)
 ndarray(shape, dtype=float, buffer=None, offset=0,
 strides=None, order=None)

```

An array [object](#) represents a multidimensional, homogeneous array of fixed-size items. An associated data-type [object](#) describes the format of each element in the array (its [byte](#)-order, how many bytes it occupies in memory, whether it is an [integer](#), a [floating](#) point [number](#), or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer to the See Also section below). The parameters given here refer to a low-level method (`[ndarray](#)(...)`) for instantiating an array.

For more information, refer to the `numpy` module and examine the the methods and attributes of an array.

```
Parameters

(for the __new__ method; see Notes below)

shape : tuple of ints
 Shape of created array.
dtype : data-type, optional
 Any object that can be interpreted as a numpy data type.
buffer : object exposing buffer interface, optional
 Used to fill the array with data.
offset : int, optional
 Offset of array data in buffer.
strides : tuple of ints, optional
 Strides of data in memory.
order : {'C', 'F'}, optional
 Row-major (C-style) or column-major (Fortran-style) order.

Attributes

T : ndarray
 Transpose of the array.
data : buffer
 The array's elements, in memory.
dtype : dtype object
 Describes the format of the elements in the array.
flags : dict
 Dictionary containing information related to memory use, e.g.,
 'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', etc.
flat : numpy.flatiter object
 Flattened version of the array as an iterator. The iterator
 allows assignments, e.g., ``x.flat = 3`` (See `ndarray.flat` for
 assignment examples; TODO).
imag : ndarray
 Imaginary part of the array.
real : ndarray
 Real part of the array.
size : int
 Number of elements in the array.
itemsize : int
 The memory use of each array element in bytes.
nbytes : int
 The total number of bytes required to store the array data,
 i.e., ``itemsize * size``.
ndim : int
 The array's number of dimensions.
shape : tuple of ints
 Shape of the array.
strides : tuple of ints
 The step-size required to move from one element to the next in
 memory. For example, a contiguous ``(3, 4)`` array of type
 int16 in C-order has strides ``(8, 2)``. This implies that
 to move from element to element in memory requires jumps of 2 bytes.
 To move from row-to-row, one needs to jump 8 bytes at a time
 ``(``2 * 4``)``.
ctypes : ctypes object
 Class containing properties of the array needed for interaction
 with ctypes.
base : ndarray
 If the array is a view into another array, that array is its `base`
 (unless that array is also a view). The `base` array is where the
 array data is actually stored.

See Also

array : Construct an array.
zeros : Create an array, each element of which is zero.
empty : Create an array, but leave its allocated memory unchanged (i.e.,
 it contains "garbage").
dtype : Create a data-type.

Notes

There are two modes of creating an array using ``__new__``:
1. If `buffer` is None, then only `shape`, `dtype`, and `order`
 are used.
2. If `buffer` is an object exposing the buffer interface, then
 all keywords are interpreted.

No ``__init__`` method is needed because the array is fully initialized
after the ``__new__`` method.

Examples

These examples illustrate the low-level `ndarray` constructor. Refer
to the 'See Also' section above for easier ways of constructing an
ndarray.

First mode, `buffer` is None:
```

```
>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[-1.13698227e+002, 4.25087011e-303],
 [2.88528414e-306, 3.27025015e-309]]) #random

Second mode:

>>> np.ndarray((2,), buffer=np.array([1,2,3]),
... offset=np.int_().itemsize,
... dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
```

Methods defined here:

**abs**(...)  
 $x.\underline{\text{abs}}() \iff \text{abs}(x)$

**add**(...)  
 $x.\underline{\text{add}}(y) \iff x+y$

**and**(...)  
 $x.\underline{\text{and}}(y) \iff x\&y$

**array**(...)  
 $a.\underline{\text{array}}(\text{dtype}) \rightarrow \text{reference if type unchanged, copy otherwise.}$   
 Returns either a new reference to self if **dtype** is not given or a new array of provided data type if **dtype** is different from the current **dtype** of the array.

**array\_prepare**(...)  
 $a.\underline{\text{array\_prepare}}(\text{obj}) \rightarrow \text{Object of same type as ndarray object obj.}$

**array\_wrap**(...)  
 $a.\underline{\text{array\_wrap}}(\text{obj}) \rightarrow \text{Object of same type as ndarray object a.}$

**contains**(...)  
 $x.\underline{\text{contains}}(y) \iff y \text{ in } x$

**copy**(...)  
 $a.\underline{\text{copy}}([\text{order}])$   
 Return a copy of the array.  
 Parameters  
 -----  
 $\text{order} : \{\text{'C'}, \text{'F'}, \text{'A'}\}, \text{optional}$   
     If order is 'C' (False) then the result is contiguous (default).  
     If order is 'Fortran' (True) then the result has fortran order.  
     If order is 'Any' (None) then the result has fortran order  
     only if the array already is in fortran order.

**deepcopy**(...)  
 $a.\underline{\text{deepcopy}}() \rightarrow \text{Deep copy of array.}$   
 Used if copy.deepcopy is called on an array.

**delitem**(...)  
 $x.\underline{\text{delitem}}(y) \iff \text{del } x[y]$

**delslice**(...)  
 $x.\underline{\text{delslice}}(i, j) \iff \text{del } x[i:j]$   
 Use of negative indices is not supported.

**div**(...)  
 $x.\underline{\text{div}}(y) \iff x/y$

**divmod**(...)  
 $x.\underline{\text{divmod}}(y) \iff \text{divmod}(x, y)$

**eq**(...)  
 $x.\underline{\text{eq}}(y) \iff x==y$

**float**(...)  
 $x.\underline{\text{float}}() \iff \text{float}(x)$

**floordiv**(...)  
 $x.\underline{\text{floordiv}}(y) \iff x//y$

```
__ge__(...)
x.__ge__(y) <==> x>=y

__getitem__(...)
x.__getitem__(y) <==> x[y]

__getslice__(...)
x.__getslice__(i, j) <==> x[i:j]

 Use of negative indices is not supported.

__gt__(...)
x.__gt__(y) <==> x>y

__hex__(...)
x.__hex__() <==> hex(x)

__iadd__(...)
x.__iadd__(y) <==> x+=y

__iand__(...)
x.__iand__(y) <==> x&=y

__idiv__(...)
x.__idiv__(y) <==> x/=y

__ifloordiv__(...)
x.__ifloordiv__(y) <==> x//=y

__ilshift__(...)
x.__ilshift__(y) <==> x<<=y

__imod__(...)
x.__imod__(y) <==> x%-=y

__imul__(...)
x.__imul__(y) <==> x*=y

__index__(...)
x[y:z] <==> x[y.__index__():z.__index__()]

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__ior__(...)
x.__ior__(y) <==> x|=y

__ipow__(...)
x.__ipow__(y) <==> x**=y

__irshift__(...)
x.__irshift__(y) <==> x>>=y

__isub__(...)
x.__isub__(y) <==> x-=y

__iter__(...)
x.__iter__() <==> iter(x)

__itruediv__(...)
x.__itruediv__(y) <==> x/=y

__ixor__(...)
x.__ixor__(y) <==> x^=y

__le__(...)
x.__le__(y) <==> x<=y

__len__(...)
x.__len__() <==> len(x)
```

```
long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

lt(...)
x._lt_(y) <==> x<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

ne(...)
x._ne_(y) <==> x!=y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)
a._reduce_()

For pickling.

repr(...)
x._repr_() <==> repr(x)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

rmod(...)
x._rmod_(y) <==> y%x

rmul(...)
x._rmul_(y) <==> y*x

ror(...)
x._ror_(y) <==> y|x

rpow(...)
y._rpow_(x[, z]) <==> pow(x, y[, z])
```

```
__rrshift__(...)
 x.__rrshift__(y) <==> y>>x

__rshift__(...)
 x.__rshift__(y) <==> x>>y

__rsub__(...)
 x.__rsub__(y) <==> y-x

__rtruediv__(...)
 x.__rtruediv__(y) <==> y/x

__rxor__(...)
 x.__rxor__(y) <==> y^x

__setitem__(...)
 x.__setitem__(i, y) <==> x[i]=y

__setslice__(...)
 x.__setslice__(i, j, y) <==> x[i:j]=y

 Use of negative indices is not supported.

__setstate__(...)
 a.__setstate__(version, shape, dtype, isfortran, rawdata)

 For unpickling.

 Parameters

 version : int
 optional pickle version. If omitted defaults to 0.
 shape : tuple
 dtype : data-type
 isFortran : bool
 rawdata : string or list
 a binary string with the data (or a list if 'a' is an object array)

__sizeof__(...)

__str__(...)
 x.__str__() <==> str(x)

__sub__(...)
 x.__sub__(y) <==> x-y

__truediv__(...)
 x.__truediv__(y) <==> x/y

__xor__(...)
 x.__xor__(y) <==> x^y

all(...)
 a.all(axis=None, out=None, keepdims=False)

 Returns True if all elements evaluate to True.

 Refer to `numpy.all` for full documentation.

 See Also

 numpy.all : equivalent function

any(...)
 a.any(axis=None, out=None, keepdims=False)

 Returns True if any of the elements of `a` evaluate to True.

 Refer to `numpy.any` for full documentation.

 See Also

 numpy.any : equivalent function

argmax(...)
 a.argmax(axis=None, out=None)

 Return indices of the maximum values along the given axis.

 Refer to `numpy.argmax` for full documentation.
```

```
See Also

numpy.argmax : equivalent function

argmin(...)
a.argmin(axis=None, out=None)

 Return indices of the minimum values along the given axis of `a`.

 Refer to `numpy.argmin` for detailed documentation.

See Also

numpy.argmax : equivalent function

argpartition(...)
a.argpartition(kth, axis=-1, kind='introselect', order=None)

 Returns the indices that would partition this array.

 Refer to `numpy.argpartition` for full documentation.

.. versionadded:: 1.8.0

See Also

numpy.argpartition : equivalent function

argsort(...)
a.argsort(axis=-1, kind='quicksort', order=None)

 Returns the indices that would sort this array.

 Refer to `numpy.argsort` for full documentation.

See Also

numpy.argsort : equivalent function

astype(...)
a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)

 Copy of the array, cast to a specified type.

 Parameters

 dtype : str or dtype
 Typecode or data-type to which the array is cast.
 order : {'C', 'F', 'A', 'K'}, optional
 Controls the memory layout order of the result.
 'C' means C order, 'F' means Fortran order, 'A'
 means 'F' order if all the arrays are Fortran contiguous,
 'C' order otherwise, and 'K' means as close to the
 order the array elements appear in memory as possible.
 Default is 'K'.
 casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
 Controls what kind of data casting may occur. Defaults to 'unsafe'
 for backwards compatibility.

 * 'no' means the data types should not be cast at all.
 * 'equiv' means only byte-order changes are allowed.
 * 'safe' means only casts which can preserve values are allowed.
 * 'same_kind' means only safe casts or casts within a kind,
 like float64 to float32, are allowed.
 * 'unsafe' means any data conversions may be done.
 subok : bool, optional
 If True, then sub-classes will be passed-through (default), otherwise
 the returned array will be forced to be a base-class array.
 copy : bool, optional
 By default, astype always returns a newly allocated array. If this
 is set to false, and the 'dtype', 'order', and 'subok'

 requirements are satisfied, the input array is returned instead
 of a copy.

 Returns

 arr_t : ndarray
 Unless 'copy' is False and the other conditions for returning the input
 array are satisfied (see description for 'copy' input parameter), 'arr_t'
 is a new array of the same shape as the input array, with dtype, order
 given by 'dtype', 'order'.

 Notes

 Starting in NumPy 1.9, astype method now returns an error if the string
 dtype to cast to is not long enough in 'safe' casting mode to hold the max
```

value of `integer/float` array that is being casted. Previously the casting was allowed even if the result was truncated.

Raises

-----  
`ComplexWarning`

When casting from `complex` to `float` or `int`. To avoid this, one should use ```a.real.astype(t)```.

Examples

-----  
`>>> x = np.array([1, 2, 2.5])`  
`>>> x`  
`array([ 1., 2., 2.5])`  
`>>> x.astype(int)`  
`array([1, 2, 2])`

**byteswap(...)**

`a.byteswap(inplace)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

-----  
`inplace : bool, optional`  
If ``True``, swap bytes in-place, default is ``False``.

Returns

-----  
`out : ndarray`  
The byteswapped array. If `inplace` is ``True``, this is a view to self.

Examples

-----  
`>>> A = np.array([1, 256, 8755], dtype=np.int16)`  
`>>> map(hex, A)`  
['0x1', '0x100', '0x2233']  
`>>> A.byteswap(True)`  
`array([ 256, 1, 13090], dtype=int16)`  
`>>> map(hex, A)`  
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

`>>> A = np.array(['ceg', 'fac'])`  
`>>> A.byteswap()`  
`array(['ceg', 'fac'],`  
`dtype='|S3')`

**choose(...)**

`a.choose(choices, out=None, mode='raise')`

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also

-----  
`numpy.choose` : equivalent function

**clip(...)**

`a.clip(min=None, max=None, out=None)`

Return an array whose values are limited to ``[min, max]``.  
One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See Also

-----  
`numpy.clip` : equivalent function

**compress(...)**

`a.compress(condition, axis=None, out=None)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

-----  
`numpy.compress` : equivalent function

**conj(...)**  
a.[conj\(\)](#)

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

-----

`numpy.conjugate` : equivalent function

**conjugate(...)**  
a.[conjugate\(\)](#)

Return the [complex](#) conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See Also

-----

`numpy.conjugate` : equivalent function

**copy(...)**  
a.[copy\(order='C'\)](#)

Return a copy of the array.

Parameters

-----

order : {'C', 'F', 'A', 'K'}, optional  
Controls the memory layout of the copy. 'C' means C-order,  
'F' means F-order, 'A' means 'F' if 'a' is Fortran contiguous,  
'C' otherwise. 'K' means match the layout of 'a' as closely  
as possible. (Note that this function and :func:`numpy.copy` are very  
similar, but have different default values for their order= arguments.)

See also

-----

`numpy.copy`  
`numpy.copyto`

Examples

-----

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy\(\)
>>> x.fill(0)
>>> x
array([[0, 0, 0],
 [0, 0, 0]])
>>> y
array([[1, 2, 3],
 [4, 5, 6]])
>>> y.flags['C_CONTIGUOUS']
True
```

**cumprod(...)**  
a.[cumprod\(axis=None, dtype=None, out=None\)](#)

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also

-----

`numpy.cumprod` : equivalent function

**cumsum(...)**  
a.[cumsum\(axis=None, dtype=None, out=None\)](#)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also

-----

`numpy.cumsum` : equivalent function

**diagonal(...)**  
a.[diagonal\(offset=0, axis1=0, axis2=1\)](#)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to :func:`numpy.diagonal` for full documentation.

See Also

-----

`numpy.diagonal` : equivalent function

**dot(...)**

`a.dot(b, out=None)`

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See Also

-----

`numpy.dot` : equivalent function

Examples

-----

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2., 2.],
 [2., 2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8., 8.],
 [8., 8.]])
```

**dump(...)**

`a.dump(file)`

Dump a pickle of the array to the specified file.  
The array can be read back with `pickle.load` or `numpy.load`.

Parameters

-----

`file : str`  
A string naming the dump file.

**dumps(...)**

`a.dumps()`

Returns the pickle of the array as a string.  
`pickle.loads` or `numpy.loads` will convert the string back to an array.

Parameters

-----

`None`

**fill(...)**

`a.fill(value)`

Fill the array with a scalar value.

Parameters

-----

`value : scalar`  
All elements of `a` will be assigned this value.

Examples

-----

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```

**flatten(...)**

`a.flatten(order='C')`

Return a copy of the array collapsed into one dimension.

Parameters

-----

`order : {'C', 'F', 'A', 'K'}, optional`  
'C' means to flatten in row-major (C-style) order.

'F' means to flatten in column-major (Fortran-style) order. 'A' means to flatten in column-major order if `a` is Fortran \*contiguous\* in memory, row-major order otherwise. 'K' means to flatten `a` in the order the elements occur in memory. The default is 'C'.

Returns

y : `ndarray`  
A copy of the input array, flattened to one dimension.

See Also

ravel : Return a flattened array.  
flat : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

**getfield(...)**

a.`getfield(dtype`, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view `dtype` fits in the array `dtype`; for example an array of `dtype complex128` has 16-byte elements. If taking a view with a 32-bit `integer` (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

`dtype` : `str` or `dtype`  
The data type of the view. The `dtype` size of the view can not be larger than that of the array itself.

`offset` : `int`  
Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j, 0.+0.j],
 [0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[1., 0.],
 [0., 2.]])
```

By choosing an offset of 8 bytes we can select the `complex` part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1., 0.],
 [0., 4.]])
```

**item(...)**

a.`item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

`*args` : Arguments (variable `number` and type)

- \* `none`: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar `object` and returned.
- \* `int_type`: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- \* `tuple of int_types`: functions as does a `single` `int_type` argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar `object`  
A copy of the specified element of the array as a suitable Python scalar

**Notes**  
 When the data type of `a` is `longdouble` or `clongdouble`, `item()` returns a scalar array `object` because there is no available Python scalar that would not lose information. Void arrays return a buffer `object` for `item()`, unless fields are defined, in which case a tuple is returned.

`item` is very similar to `a[args]`, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

**Examples**

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

**itemset(...)**

```
a.itemset(*args)
```

Insert scalar into an array (scalar is cast to array's `dtype`, if possible)

There must be at least 1 argument, and define the last argument as `*args`. Then, ```a.itemset(*args)``` is equivalent to but faster than ```a[args] = item```. The item should be a scalar value and `args` must select a `single` item in the array `a`.

**Parameters**

```
args : Arguments
 If one argument: a scalar, only used in case `a` is of size 1.
 If two arguments: the last argument is the value to be set
 and must be a scalar, the first argument specifies a single array
 element location. It is either an int or a tuple.
```

**Notes**

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

**Examples**

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
 [2, 0, 3],
 [8, 5, 9]])
```

**max(...)**

```
a.max(axis=None, out=None)
```

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

**See Also**

```
numpy.amax : equivalent function
```

**mean(...)**

```
a.mean(axis=None, dtype=None, out=None, keepdims=False)
```

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

```
See Also

numpy.mean : equivalent function

min(...)
a.min(axis=None, out=None, keepdims=False)

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also

numpy.amin : equivalent function

newbyteorder(...)
arr.newbyteorder(new_order='S')

Return the array with the same data viewed with a different byte order.

Equivalent to::

 arr.view(arr.dtype.newbyteorder(new_order))

Changes are also made in all fields and sub-arrays of the array data
type.

Parameters

new_order : string, optional
 Byte order to force; a value from the byte order specifications
 below. `new_order` codes can be any of:
 * 'S' - swap dtype from current to opposite endian
 * {'<', '>'} - little endian
 * {'>', '<'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_arr : array
 New array object with the dtype reflecting given change to the
 byte order.

nonzero(...)
a.nonzero()

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See Also

numpy.nonzero : equivalent function

partition(...)
a.partition(kth, axis=-1, kind='introselect', order=None)

Rearranges the elements in the array in such a way that value of the
element in kth position is in the position it would be in a sorted array.
All elements smaller than the kth element are moved before this element and
all equal or greater are moved behind it. The ordering of the elements in
the two partitions is undefined.

.. versionadded:: 1.8.0

Parameters

kth : int or sequence of ints
 Element index to partition by. The kth element value will be in its
 final sorted position and all smaller elements will be moved before it
 and all equal or greater elements behind it.
 The order all elements in the partitions is undefined.
 If provided with a sequence of kth it will partition all elements
 indexed by kth of them into their sorted position at once.
axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
```

```
 last axis.
kind : {'introselect'}, optional
 Selection algorithm. Default is 'introselect'.
order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.partition : Return a partitioned copy of an array.
argpartition : Indirect partition.
sort : Full sort.

Notes

See ``np.partition`` for notes on the different algorithms.

Examples

>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])

prod(...)
a.prod(axis=None, dtype=None, out=None, keepdims=False)

 Return the product of the array elements over the given axis

 Refer to `numpy.prod` for full documentation.

See Also

numpy.prod : equivalent function

ptp(...)
a.ptp(axis=None, out=None)

 Peak to peak (maximum - minimum) value along a given axis.

 Refer to `numpy.ptp` for full documentation.

See Also

numpy.ptp : equivalent function

put(...)
a.put(indices, values, mode='raise')

 Set ``a.flat[n] = values[n]`` for all `n` in indices.

 Refer to `numpy.put` for full documentation.

See Also

numpy.put : equivalent function

ravel(...)
a.ravel([order])

 Return a flattened array.

 Refer to `numpy.ravel` for full documentation.

See Also

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat(...)
a.repeat(repeats, axis=None)

 Repeat elements of an array.

 Refer to `numpy.repeat` for full documentation.

See Also

numpy.repeat : equivalent function
```

```
reshape(...)
a.reshape(shape, order='C')

 Returns an array containing the same data with a new shape.

 Refer to `numpy.reshape` for full documentation.

See Also

numpy.reshape : equivalent function

resize(...)
a.resize(new_shape, refcheck=True)

 Change shape and size of array in-place.

Parameters

new_shape : tuple of ints, or 'n' ints
 Shape of resized array.
refcheck : bool, optional
 If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError
 If `a` does not own its own data or references or views to it exist,
 and the data memory must be changed.

SystemError
 If the 'order' keyword argument is specified. This behaviour is a
 bug in NumPy.

See Also

resize : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be
resized.

The purpose of the reference count check is to make sure you
do not use this array as a buffer for another Python object and then
reallocate the memory. However, reference counts can increase in
other ways so if you are sure that you have not shared the memory
for this array with another Python object, then you may safely set
'refcheck' to False.

Examples

Shrinking an array: array is flattened (in the order that the data are
stored in memory), resized, and reshaped:

>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
[1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
[2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
[3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...

Unless 'refcheck' is False:
```

```

>>> a.resize((1, 1), refcheck=False)
>>> a
([[0]])
>>> c
([[0]])

round(...)
a.round(decimals=0, out=None)

 Return `a` with each element rounded to the given number of decimals.

 Refer to `numpy.around` for full documentation.

 See Also

 numpy.around : equivalent function

searchsorted(...)
a.searchsorted(v, side='left', sorter=None)

 Find indices where elements of v should be inserted in a to maintain order.

 For full documentation, see `numpy.searchsorted`

 See Also

 numpy.searchsorted : equivalent function

setfield(...)
a.setfield(val, dtype, offset=0)

 Put a value into a specified place in a field defined by a data-type.

 Place `val` into `a`'s field defined by `dtype` and beginning `offset` bytes into the field.

 Parameters

 val : object
 Value to be placed in field.
 dtype : dtype object
 Data-type of the field in which to place `val`.
 offset : int, optional
 The number of bytes into the field at which to place `val`.

 Returns

 None

 See Also

 getfield

 Examples

 >>> x = np.eye(3)
 >>> x.getfield(np.float64)
([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
 >>> x.setfield(3, np.int32)
 >>> x.getfield(np.int32)
([[3, 3, 3],
 [3, 3, 3],
 [3, 3, 3]])
 >>> x
([[1.0000000e+000, 1.48219694e-323, 1.48219694e-323],
 [-1.48219694e-323, 1.0000000e+000, 1.48219694e-323],
 [-1.48219694e-323, 1.48219694e-323, 1.0000000e+000]])
 >>> x.setfield(np.eye(3), np.int32)
 >>> x
([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])

```

**setflags**(...)
a.setflags(write=None, align=None, uic=None)

 Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

 These Boolean-valued flags affect how numpy interprets the memory area used by `a` (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface,

```
or is a string. (The exception for string is made so that unpickling
can be done without copying memory.)

Parameters

write : bool, optional
 Describes whether or not `a` can be written to.
align : bool, optional
 Describes whether or not `a` is aligned properly for its type.
uic : bool, optional
 Describes whether or not `a` is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used
for the array is to be interpreted. There are 6 Boolean flags
in use, only three of which can be changed by the user:
UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware
(as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced
by .base). When this array is deallocated, the base array will be
updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well
as the full name.

Examples

>>> y
array([[3, 1, 7],
 [2, 0, 0],
 [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True

sort(...)
a.sort(axis=-1, kind='quicksort', order=None)

Sort an array, in-place.

Parameters

axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 Sorting algorithm. Default is 'quicksort'.
order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.sort : Return a sorted copy of an array.
argsort : Indirect sort.
lexsort : Indirect stable sort on multiple keys.
searchsorted : Find elements in sorted array.
partition: Partial sort.

Notes

See ``sort`` for notes on the different sorting algorithms.

Examples
```

```

>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
 [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
 [1, 4]])

Use the `order` keyword to specify a field to use when sorting a
structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
 dtype=[('x', '|S1'), ('y', '<i4')])

squeeze(...)
a.squeeze(axis=None)

Remove single-dimensional entries from the shape of `a`.

Refer to `numpy.squeeze` for full documentation.

See Also

numpy.squeeze : equivalent function

std(...)
a.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also

numpy.std : equivalent function

sum(...)
a.sum(axis=None, dtype=None, out=None, keepdims=False)

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also

numpy.sum : equivalent function

swapaxes(...)
a.swapaxes(axis1, axis2)

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

numpy.swapaxes : equivalent function

take(...)
a.take(indices, axis=None, out=None, mode='raise')

Return an array formed from the elements of `a` at the given indices.

Refer to `numpy.take` for full documentation.

See Also

numpy.take : equivalent function

tobytes(...)
a.tobytes(order='C')

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of
data memory. The bytes object can be produced in either 'C' or 'Fortran',
or 'Any' order (the default is 'C'-order). 'Any' order means C-order
unless the F_CONTIGUOUS flag in the array is set, in which case it
means 'Fortran' order.
```

```
.. versionadded:: 1.9.0

Parameters

order : {'C', 'F', None}, optional
 Order of the data for multidimensional arrays:
 C, Fortran, or the same as for the original array.

Returns

s : bytes
 Python bytes exhibiting a copy of `a`'s raw data.

Examples

>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x02\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'

tofile(...)
a.tofile(fid, sep="", format="%s")

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of `a`.
The data produced by this method can be recovered using the function
fromfile\(\).

Parameters

fid : file or str
 An open file object, or a string containing a filename.
sep : str
 Separator between array items for text output.
 If "" (empty), a binary file is written, equivalent to
 ``file.write(a.tobytes())``.
format : str
 Format string for text file output.
 Each entry in the array is formatted to text by first converting
 it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data.
Information on endianness and precision is lost, so this method is not a
good choice for files intended to archive data or transport data between
machines with different endianness. Some of these problems can be overcome
by outputting the data as text files, at the expense of speed and file
size.

tolist(...)
a.tolist()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list.
Data items are converted to the nearest compatible Python type.

Parameters

none

Returns

y : list
 The possibly nested list of array elements.

Notes

The array may be recreated, ``a = np.array(a.tolist())``.

Examples

>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]

tostring(...)
a.tostring(order='C')
```

```
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of
data memory. The bytes object can be produced in either 'C' or 'Fortran',
or 'Any' order (the default is 'C'-order). 'Any' order means C-order
unless the F_CONTIGUOUS flag in the array is set, in which case it
means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

Parameters

order : {'C', 'F', None}, optional
 Order of the data for multidimensional arrays:
 C, Fortran, or the same as for the original array.

Returns

s : bytes
 Python bytes exhibiting a copy of `a`'s raw data.

Examples

>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x02\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'

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

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also

numpy.trace : equivalent function

transpose(...)
a.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and
row vectors, first cast the 1-D array into a matrix object.)

For a 2-D array, this is the usual matrix transpose.

For an n-D array, if axes are given, their order indicates how the
axes are permuted (see Examples). If axes are not provided and
``a.shape = (i[0], i[1], ... i[n-2], i[n-1])``, then
``a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])``.

Parameters

axes : None, tuple of ints, or `n` ints
 * None or no argument: reverses the order of the axes.
 * tuple of ints: `i` in the `j`-th place in the tuple means `a`'s
 `i`-th axis becomes `a.transpose()`'s `j`-th axis.
 * `n` ints: same as an n-tuple of the same ints (this form is
 intended simply as a "convenience" alternative to the tuple form)

Returns

out : ndarray
 View of `a`, with axes suitably permuted.

See Also

ndarray.T : Array property returning the array transposed.

Examples

>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
 [3, 4]])
>>> a.transpose()
array([[1, 3],
 [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
```

```

[2, 4])
>>> a.transpose(1, 0)
array([[1, 3],
 [2, 4]])

var(...)
a.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

 Returns the variance of the array elements, along given axis.

 Refer to `numpy.var` for full documentation.

See Also

numpy.var : equivalent function

view(...)
a.view(dtype=None, type=None)

 New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional
 Data-type descriptor of the returned view, e.g., float32 or int16. The
 default, None, results in the view having the same data-type as `a`.
 This argument can also be specified as an ndarray sub-class, which
 then specifies the type of the returned object (this is equivalent to
 setting the ``type`` parameter).
type : Python type, optional
 Type of the returned view, e.g., ndarray or matrix. Again, the
 default None results in type preservation.

Notes

``a.view()`` is used two different ways:

``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a view
of the array's memory with a different data-type. This can cause a
reinterpretation of the bytes of memory.

``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` just
returns an instance of `ndarray_subclass` that looks at the same array
(same shape, dtype, etc.) This does not cause a reinterpretation of the
memory.

For ``a.view(some_dtype)``, if ``some_dtype`` has a different number of
bytes per entry than the previous dtype (for example, converting a
regular array to a structured array), then the behavior of the view
cannot be predicted just from the superficial appearance of ``a`` (shown
by ``print(a)``). It also depends on exactly how ``a`` is stored in
memory. Therefore if ``a`` is C-ordered versus fortran-ordered, versus
defined as a slice or transpose, etc., the view may give different
results.

Examples

>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
Viewing array data using a different type and dtype:

>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.matrix'>

Creating a view on a structured array so it can be used in calculations

>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
 [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])

Making changes to the view changes the underlying array

>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]

Using a view to convert an array to a recarray:

>>> z = x.view(np.recarray)
>>> z.a

```

```
array([1], dtype=int8)
Views share data:
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)

Views that change the dtype size (bytes per entry) should normally be
avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
 [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([(1, 2),
 (4, 5)], dtype=[('width', '<i2'), ('length', '<i2')])
```

---

Data descriptors defined here:

## T

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

Examples

```

>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[1., 2.],
 [3., 4.]])
>>> x.T
array([[1., 3.],
 [2., 4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([1., 2., 3., 4.])
>>> x.T
array([1., 2., 3., 4.])
```

## \_array\_finalize\_

None.

## \_array\_interface\_

Array protocol: Python side.

## \_array\_priority\_

Array priority.

## \_array\_struct\_

Array protocol: C-struct side.

## base

Base object if memory is from some other object.

Examples

-----

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

## ctypes

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

```

Parameters

None

Returns

c : Python object
 Possessing attributes data, shape, strides, etc.

See Also

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented
in "Guide to NumPy" (we have omitted undocumented public attributes,
as well as documented private attributes):

* data: A pointer to the memory area of the array as a Python integer.
 This memory area may contain data that is not aligned, or not in correct
 byte-order. The memory area may not even be writeable. The array
 flags and data-type of this array should be respected when passing this
 attribute to arbitrary C-code to avoid trouble that can include Python
 crashing. User Beware! The value of this attribute is exactly the same
 as self._array_interface_['data'][0].

* shape (c_intp*self.ndim): A ctypes array of length self.ndim where
 the basetype is the C-integer corresponding to dtype('p') on this
 platform. This base-type could be c_int, c_long, or c_longlong
 depending on the platform. The c_intp type is defined accordingly in
 numpy.ctypeslib. The ctypes array contains the shape of the underlying
 array.

* strides (c_intp*self.ndim): A ctypes array of length self.ndim where
 the basetype is the same as for the shape attribute. This ctypes array
 contains the strides information from the underlying array. This strides
 information is important for showing how many bytes must be jumped to
 get to the next element in the array.

* data_as(obj): Return the data pointer cast to a particular c-types object.
 For example, calling self._as_parameter_ is equivalent to
 self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a
 pointer to a ctypes array of floating-point data:
 self.data_as(ctypes.POINTER(ctypes.c_double)).

* shape_as(obj): Return the shape tuple as an array of some other c-types
 type. For example: self.shape_as(ctypes.c_short).

* strides_as(obj): Return the strides tuple as an array of some other
 c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary
arrays or arrays constructed on the fly. For example, calling
``(a+b).ctypes.data_as(ctypes.c_void_p)`` returns a pointer to memory
that is invalid because the array created as (a+b) is deallocated
before the next Python statement. You can avoid this problem using
either ``c=a+b`` or ``ct=(a+b).ctypes``. In the latter case, ct will
hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute
of array objects still returns something useful, but ctypes objects
are not returned and errors may be raised instead. In particular,
the object will still have the as parameter attribute which will
return an integer equal to the data attribute.

Examples

>>> import ctypes
>>> x
array([[0, 1],
 [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>

```

**data**

Python buffer object pointing to the start of the array's data.

**dtype**

Data-type of the array's elements.

Parameters

-----

None

Returns

-----

d : numpy dtype object

See Also

-----

`numpy.dtype`

Examples

-----

```
>>> x
array([[0, 1],
 [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

**flags**

Information about the memory layout of the array.

Attributes

-----

C\_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F\_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

UPDATEIFCOPY (U)

This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

FNC

F\_CONTIGUOUS and not C\_CONTIGUOUS.

FORC

F\_CONTIGUOUS or C\_CONTIGUOUS (one-segment test).

BEHAVED (B)

ALIGNED and WRITEABLE.

CARRAY (CA)

BEHAVED and C\_CONTIGUOUS.

FARRAY (FA)

BEHAVED and F\_CONTIGUOUS and not C\_CONTIGUOUS.

Notes

-----

The `flags` object can be accessed dictionary-like (as in ``a.flags['WRITEABLE']``), or by using lowercased attribute names (as in ``a.flags.writeable``). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set ``False``.
- ALIGNED can only be set ``True`` if the data is truly aligned.
- WRITEABLE can only be set ``True`` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension ``arr.strides[dim]`` may be \*arbitrary\* if ``arr.shape[dim] == 1`` or the array has no elements. It does \*not\* generally hold that ``self.strides[-1] == self.itemsize`` for C-style contiguous arrays or ``self.strides[0] == self.itemsize`` for Fortran-style contiguous arrays is true.

**flat**

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

-----

`flatten` : Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
 [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
 [2, 5],
 [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
 [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
 [3, 1, 3]])
```

**imag**

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

**itemsize**

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

**nbytes**

Total bytes consumed by the elements of the array.

Notes

-----  
Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**ndim**

```

Number of array dimensions.

Examples

>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3

real
The real part of the array.

Examples

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([1. , 0.70710678])
>>> x.real.dtype
dtype('float64')

See Also

numpy.real : equivalent function

shape
Tuple of array dimensions.

Notes

May be used to "reshape" the array, as long as this would not
require a change in the total number of elements

Examples

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged

size
Number of elements in the array.

Equivalent to ``np.prod(a.shape)``, i.e., the product of the array's
dimensions.

Examples

>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30

strides
Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ``(i[0], i[1], ..., i[n])`` in an array `a` is::

 offset = sum(np.array(i) * a.strides)

A more detailed explanation of strides can be found in the
"ndarray.rst" file in the NumPy reference guide.

Notes

Imagine an array of 32-bit integers (each 4 bytes)::

 x = np.array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]], dtype=np.int32)

This array is stored in memory as 40 bytes, one after the other
(known as a contiguous block of memory). The strides of an array tell
us how many bytes we have to skip in memory to move to the next position

```

along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be `(20, 4)`.

See Also

-----

[numpy.lib.stride\\_tricks.as\\_strided](#)

Examples

-----

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]],
 [[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Data and other attributes defined here:

**\_\_hash\_\_** = None

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

### class ndenumerate(builtin \_\_object)

Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

Parameters

-----

**arr** : ndarray  
Input array.

See Also

-----

[ndindex](#), [flatiter](#)

Examples

-----

```
>>> a = np.array([[1, 2], [3, 4]])
>>> for index, x in np.ndenumerate(a):
... print(index, x)
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

Methods defined here:

**\_\_init\_\_**(self, arr)

**\_\_iter\_\_**(self)

**\_\_next\_\_**(self)

Standard iterator method, returns the index tuple and array value.

Returns

-----

**coords** : tuple of ints  
The indices of the current iteration.  
**val** : scalar

The array element of the current iteration.

**next** = [\\_next\\_\(self\)](#)

---

Data descriptors defined here:

**\_dict\_**  
dictionary for instance variables (if defined)

**\_weakref\_**  
list of weak references to the object (if defined)

**class ndindex([builtin\\_.object](#))**  
An N-dimensional iterator [object](#) to index arrays.

Given the shape of an array, an `ndindex` instance iterates over the N-dimensional index of the array. At each iteration a tuple of indices is returned, the last dimension is iterated over first.

Parameters  
-----  
`\*args` : ints  
The size of each dimension of the array.

See Also  
-----  
[ndenumerate](#), [flatiter](#)

Examples  
-----  

```
>>> for index in np.ndindex(3, 2, 1):
... print(index)
(0, 0, 0)
(0, 1, 0)
(1, 0, 0)
(1, 1, 0)
(2, 0, 0)
(2, 1, 0)
```

Methods defined here:

**\_init\_(self, \*shape)**

**\_iter\_(self)**

**\_next\_(self)**  
Standard iterator method, updates the index and returns the index tuple.

Returns  
-----  
val : tuple of ints  
Returns a tuple containing the indices of the current iteration.

**ndincr(self)**  
Increment the multi-dimensional index by one.

This method is for backward compatibility only: do not use.

**next** = [\\_next\\_\(self\)](#)

---

Data descriptors defined here:

**\_dict\_**  
dictionary for instance variables (if defined)

**\_weakref\_**  
list of weak references to the object (if defined)

**class nditer([builtin\\_.object](#))**  
Efficient multi-dimensional iterator [object](#) to iterate over arrays.  
To get started using this [object](#), see the  
:ref:`introductory guide to array iteration <arrays.nditer>`.

**Parameters**

-----

**op** : [ndarray](#) or sequence of [array\\_like](#)  
 The [array](#)(s) to iterate over.

**flags** : sequence of [str](#), optional  
 Flags to control the behavior of the iterator.

- \* "buffered" enables buffering when required.
- \* "c\_index" causes a C-order index to be tracked.
- \* "f\_index" causes a Fortran-order index to be tracked.
- \* "multi\_index" causes a multi-index, or a tuple of indices with one per iteration dimension, to be tracked.
- \* "common\_dtype" causes all the operands to be converted to a common data type, with copying or buffering as necessary.
- \* "delay\_bufalloc" delays allocation of the buffers until a [reset\(\)](#) call is made. Allows "allocate" operands to be initialized before their values are copied into the buffers.
- \* "external\_loop" causes the `values` given to be one-dimensional arrays with multiple values instead of zero-dimensional arrays.
- \* "grow\_inner" allows the `value` array sizes to be made larger than the buffer size when both "buffered" and "external\_loop" is used.
- \* "ranged" allows the iterator to be restricted to a sub-range of the iterindex values.
- \* "refs\_ok" enables iteration of reference types, such as [object](#) arrays.
- \* "reduce\_ok" enables iteration of "readwrite" operands which are broadcasted, also known as reduction operands.
- \* "zerosize\_ok" allows `itersize` to be zero.

**op\_flags** : list of list of [str](#), optional  
 This is a list of flags for each operand. At minimum, one of "readonly", "readwrite", or "writeonly" must be specified.

- \* "readonly" indicates the operand will only be read from.
- \* "readwrite" indicates the operand will be read from and written to.
- \* "writeonly" indicates the operand will only be written to.
- \* "no\_broadcast" prevents the operand from being broadcasted.
- \* "contig" forces the operand data to be contiguous.
- \* "aligned" forces the operand data to be aligned.
- \* "nbo" forces the operand data to be in native [byte](#) order.
- \* "copy" allows a temporary read-only copy if required.
- \* "updateifcopy" allows a temporary read-write copy if required.
- \* "allocate" causes the array to be allocated if it is None in the `op` parameter.
- \* "no\_subtype" prevents an "allocate" operand from using a subtype.
- \* "arraymask" indicates that this operand is the mask to use for selecting elements when writing to operands with the 'writemasked' flag set. The iterator does not enforce this, but when writing from a buffer back to the array, it only copies those elements indicated by this mask.
- \* 'writemasked' indicates that only elements where the chosen 'arraymask' operand is True will be written to.

**op\_dtypes** : [dtype](#) or tuple of [dtype](#)(s), optional  
 The required data type(s) of the operands. If copying or buffering is enabled, the data will be converted to/from their original types.

**order** : {'C', 'F', 'A', 'K'}, optional  
 Controls the iteration order. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. This also affects the element memory order of "allocate" operands, as they are allocated to be compatible with iteration order.  
 Default is 'K'.

**casting** : {'no', 'equiv', 'safe', 'same\_kind', 'unsafe'}, optional  
 Controls what kind of data casting may occur when making a copy or buffering. Setting this to 'unsafe' is not recommended, as it can adversely affect accumulations.

- \* 'no' means the data types should not be cast at all.
- \* 'equiv' means only [byte](#)-order changes are allowed.
- \* 'safe' means only casts which can preserve values are allowed.
- \* 'same\_kind' means only safe casts or casts within a kind, like [float64](#) to [float32](#), are allowed.
- \* 'unsafe' means any data conversions may be done.

**op\_axes** : list of list of ints, optional  
 If provided, is a list of ints or None for each operands.  
 The list of axes for an operand is a mapping from the dimensions of the iterator to the dimensions of the operand. A value of -1 can be placed for entries, causing that dimension to be treated as "newaxis".

**itershape** : tuple of ints, optional  
 The desired shape of the iterator. This allows "allocate" operands with a dimension mapped by op\_axes not corresponding to a dimension of a different operand to get a value not equal to 1 for that dimension.

**buffersize** : [int](#), optional  
 When buffering is enabled, controls the size of the temporary

```

buffers. Set to 0 for the default value.

Attributes

dtypes : tuple of dtype(s)
 The data types of the values provided in `value`. This may be
 different from the operand data types if buffering is enabled.
finished : bool
 Whether the iteration over the operands is finished or not.
has_delayed_bufalloc : bool
 If True, the iterator was created with the "delay_bufalloc" flag,
 and no reset\(\) function was called on it yet.
has_index : bool
 If True, the iterator was created with either the "c_index" or
 the "f_index" flag, and the property `index` can be used to
 retrieve it.
has_multi_index : bool
 If True, the iterator was created with the "multi_index" flag,
 and the property `multi_index` can be used to retrieve it.
index :
 When the "c_index" or "f_index" flag was used, this property
 provides access to the index. Raises a ValueError if accessed
 and `has_index` is False.
iterationneedsapi : bool
 Whether iteration requires access to the Python API, for example
 if one of the operands is an object array.
iterindex : int
 An index which matches the order of iteration.
itersize : int
 Size of the iterator.
itviews :
 Structured view(s) of `operands` in memory, matching the reordered
 and optimized iterator access pattern.
multi_index :
 When the "multi_index" flag was used, this property
 provides access to the index. Raises a ValueError if accessed
 and `has_multi_index` is False.
ndim : int
 The iterator's dimension.
nop : int
 The number of iterator operands.
operands : tuple of operand(s)
 The array(s) to be iterated over.
shape : tuple of ints
 Shape tuple, the shape of the iterator.
value :
 Value of `operands` at current iteration. Normally, this is a
 tuple of array scalars, but if the flag "external_loop" is used,
 it is a tuple of one dimensional arrays.

```

#### Notes

[`nditer`](#) supersedes [`flatiter`](#). The iterator implementation behind  
[`nditer`](#) is also exposed by the Numpy C API.

The Python exposure supplies two iteration interfaces, one which follows  
the Python iterator protocol, and another which mirrors the C-style  
do-while pattern. The native Python approach is better in most cases, but  
if you need the iterator's coordinates or index, use the C-style pattern.

#### Examples

Here is how we might write an ``iter\_add`` function, using the  
Python iterator protocol:::

```

def iter_add_py(x, y, out=None):
 addop = np.add
 it = np.nditer([x, y, out], [], [
 ['readonly'], ['readonly'], ['writeonly','allocate']])
 for (a, b, c) in it:
 addop(a, b, out=c)
 return it.operands[2]

```

Here is the same function, but following the C-style pattern:::

```

def iter_add(x, y, out=None):
 addop = np.add

 it = np.nditer([x, y, out], [],
 [['readonly'], ['readonly'], ['writeonly','allocate']])

 while not it.finished:
 addop(it[0], it[1], out=it[2])
 it.iternext()

 return it.operands[2]

```

Here is an example outer product function::

```

def outer_it(x, y, out=None):
 mulop = np.multiply

 it = np.nditer([x, y, out], ['external_loop'],
 [['readonly'], ['readonly'], ['writeonly', 'allocate']],
 op_axes=[range(x.ndim)+[-1]*y.ndim,
 [-1]*x.ndim+range(y.ndim),
 None])

 for (a, b, c) in it:
 mulop(a, b, out=c)

 return it.operands[2]

>>> a = np.arange(2)+1
>>> b = np.arange(3)+1
>>> outer_it(a,b)
array([[1, 2, 3],
 [2, 4, 6]])

```

Here is an example function which operates like a "lambda" [ufunc](#):

```

def luf(lamdaexpr, *args, **kwargs):
 "luf(lamdaexpr, op1, ..., opn, out=None, order='K', casting='safe', buffersize=0)"
 nargs = len(args)
 op = (kwargs.get('out',None),) + args
 it = np.nditer(op, ['buffered','external_loop'],
 [['writeonly','allocate','no_broadcast']] +
 [['readonly','nbo','aligned']]*nargs,
 order=kwargs.get('order','K'),
 casting=kwargs.get('casting','safe'),
 buffersize=kwargs.get('buffersize',0))
 while not it.finished:
 it[0] = lamdaexpr(*it[1:])
 it.iternext()
 return it.operands[0]

>>> a = np.arange(5)
>>> b = np.ones(5)
>>> luf(lambda i,j:i*i + j/2, a, b)
array([0.5, 1.5, 4.5, 9.5, 16.5])

```

Methods defined here:

- copy**(...)
  - x.[copy](#)(y) <==> x[y]
- delitem**(...)
  - x.[delitem](#)(y) <==> del x[y]
- delslice**(...)
  - x.[delslice](#)(i, j) <==> del x[i:j]

Use of negative indices is not supported.
- getitem**(...)
  - x.[getitem](#)(y) <==> x[y]
- getslice**(...)
  - x.[getslice](#)(i, j) <==> x[i:j]

Use of negative indices is not supported.
- init**(...)
  - x.[init](#)(...) initializes x; see help(type(x)) for signature
- iter**(...)
  - x.[iter](#)() <==> iter(x)
- len**(...)
  - x.[len](#)() <==> len(x)
- setitem**(...)
  - x.[setitem](#)(i, y) <==> x[i]=y
- selslice**(...)
  - x.[selslice](#)(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.
- copy**(...)
  - [copy](#)()

```
Get a copy of the iterator in its current state.

Examples

>>> x = np.arange(10)
>>> y = x + 1
>>> it = np.nditer([x, y])
>>> it.next()
(array(0), array(1))
>>> it2 = it.copy()
>>> it2.next()
(array(1), array(2))

debug_print(...)
debug_print()

Print the current state of the `nditer` instance and debug info to stdout.

enable_external_loop(...)
enable_external_loop()

When the "external_loop" was not used during construction, but
is desired, this modifies the iterator to behave as if the flag
was specified.

iternext(...)
iternext()

Check whether iterations are left, and perform a single internal iteration
without returning the result. Used in the C-style pattern do-while
pattern. For an example, see `nditer`.

>Returns

iternext : bool
 Whether or not there are iterations left.

next(...)
x.next() -> the next value, or raise StopIteration

remove_axis(...)
remove_axis(i)

Removes axis `i` from the iterator. Requires that the flag "multi_index"
be enabled.

remove_multi_index(...)
remove_multi_index()

When the "multi_index" flag was specified, this removes it, allowing
the internal iteration structure to be optimized further.

reset(...)
reset()

Reset the iterator to its initial state.
```

---

Data descriptors defined here:

**dtypes**  
**finished**  
**has\_delayed\_bufalloc**  
**has\_index**  
**has\_multi\_index**  
**index**  
**iterationneedsapi**  
**iterindex**  
**iterrange**  
**itersize**  
**itviews**

**multi\_index****ndim****nop****operands****shape****value**

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method **\_\_new\_\_** of type object>  
T.**\_\_new\_\_**(S, ...) -> a new [object](#) with type S, a subtype of Tclass **number**([generic](#))

Method resolution order:

[number](#)  
[generic](#)  
[builtin\\_object](#)Methods inherited from [generic](#):**\_\_abs\_\_(...)**  
x.**\_\_abs\_\_**() <==> abs(x)  
  
**\_\_add\_\_(...)**  
x.**\_\_add\_\_**(y) <==> x+y  
  
**\_\_and\_\_(...)**  
x.**\_\_and\_\_**(y) <==> x&y  
  
**\_\_array\_\_(...)**  
sc.**\_\_array\_\_**(|type) return 0-dim array  
  
**\_\_array\_wrap\_\_(...)**  
sc.**\_\_array\_wrap\_\_**(obj) return scalar from array  
  
**\_\_copy\_\_(...)**  
  
**\_\_deepcopy\_\_(...)**  
  
**\_\_div\_\_(...)**  
x.**\_\_div\_\_**(y) <==> x/y  
  
**\_\_divmod\_\_(...)**  
x.**\_\_divmod\_\_**(y) <==> divmod(x, y)  
  
**\_\_eq\_\_(...)**  
x.**\_\_eq\_\_**(y) <==> x==y  
  
**\_\_float\_\_(...)**  
x.**\_\_float\_\_**() <==> [float](#)(x)  
  
**\_\_floordiv\_\_(...)**  
x.**\_\_floordiv\_\_**(y) <==> x//y  
  
**\_\_format\_\_(...)**  
NumPy array scalar formatter  
  
**\_\_ge\_\_(...)**  
x.**\_\_ge\_\_**(y) <==> x>=y  
  
**\_\_getitem\_\_(...)**  
x.**\_\_getitem\_\_**(y) <==> x[y]  
  
**\_\_gt\_\_(...)**  
x.**\_\_gt\_\_**(y) <==> x>y

```
hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

le(...)
x._le_(y) <==> x<=y

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

lt(...)
x._lt_(y) <==> x<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

ne(...)
x._ne_(y) <==> x!=y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

repr(...)
x._repr_() <==> repr(x)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

rmod(...)
x._rmod_(y) <==> y%x
```

**\_\_rmul\_\_(...)**  
x. \_\_rmul\_\_(y) <==> y\*x

**\_\_ror\_\_(...)**  
x. \_\_ror\_\_(y) <==> y|x

**\_\_rpow\_\_(...)**  
y. \_\_rpow\_\_(x[, z]) <==> pow(x, y[, z])

**\_\_rrshift\_\_(...)**  
x. \_\_rrshift\_\_(y) <==> y>>x

**\_\_rshift\_\_(...)**  
x. \_\_rshift\_\_(y) <==> x>>y

**\_\_rsub\_\_(...)**  
x. \_\_rsub\_\_(y) <==> y-x

**\_\_rtruediv\_\_(...)**  
x. \_\_rtruediv\_\_(y) <==> y/x

**\_\_rxor\_\_(...)**  
x. \_\_rxor\_\_(y) <==> y^x

**\_\_setstate\_\_(...)**

**\_\_sizeof\_\_(...)**

**\_\_str\_\_(...)**  
x. \_\_str\_\_() <==> str(x)

**\_\_sub\_\_(...)**  
x. \_\_sub\_\_(y) <==> x-y

**\_\_truediv\_\_(...)**  
x. \_\_truediv\_\_(y) <==> x/y

**\_\_xor\_\_(...)**  
x. \_\_xor\_\_(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

```

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New 'dtype' object with the given change to the byte order.

nonzero\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

```
real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

---

```
object0 = class object_(generic)
Any Python object. Character code: '0'.
```

Method resolution order:

```
object
generic
builtin .object
```

---

Methods defined here:

```
__add__(...)
 x. add_(y) <==> x+y

__call__(...)
 x. call_(...) <==> x(...)

__contains__(...)
 x. contains_(y) <==> y in x

__delattr__(...)
 x. delattr_('name') <==> del x.name

__delitem__(...)
 x. delitem_(y) <==> del x[y]

__eq__(...)
 x. eq_(y) <==> x==y

__ge__(...)
 x. ge_(y) <==> x>=y

__getattribute__(...)
 x. getattribute_('name') <==> x.name

__getitem__(...)
 x. getitem_(y) <==> x[y]

__gt__(...)
 x. gt_(y) <==> x>y

__hash__(...)
 x. hash_()
 <==> hash(x)

__iadd__(...)
 x. iadd_(y) <==> x+=y

__imul__(...)
 x. imul_(y) <==> x*=y

__le__(...)
 x. le_(y) <==> x<=y

__len__(...)
 x. len_()
 <==> len(x)

__lt__(...)
 x. lt_(y) <==> x<y

__mul__(...)
```

```

x.mul(n) <==> x*n
ne(...)
x.ne(y) <==> x!=y
rmul(...)
x.rmul(n) <==> n*x
setattr(...)
x.setattr('name', value) <==> x.name = value
setitem(...)
x.setitem(i, y) <==> x[i]=y

```

---

Data and other attributes defined here:

```

new = <built-in method new of type object>
T.new(S, ...) -> a new object with type S, a subtype of T

```

---

Methods inherited from generic:

```

abs(...)
x.abs() <==> abs(x)
and(...)
x.and(y) <==> x&y
array(...)
sc.array(|type) return 0-dim array
array_wrap(...)
sc.array_wrap(obj) return scalar from array
copy(...)
deepcopy(...)
div(...)
x.div(y) <==> x/y
divmod(...)
x.divmod(y) <==> divmod(x, y)
float(...)
x.float() <==> float(x)
floordiv(...)
x.floordiv(y) <==> x//y
format(...)
NumPy array scalar formatter
hex(...)
x.hex() <==> hex(x)
int(...)
x.int() <==> int(x)
invert(...)
x.invert() <==> ~x
long(...)
x.long() <==> long(x)
lshift(...)
x.lshift(y) <==> x<<y
mod(...)
x.mod(y) <==> x%y
neg(...)
x.neg() <==> -x

```

```
__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y
```

**\_\_xor\_\_(...)**  
x.xor(y) <=> x^y

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **conj(...)**

### **conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)****tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

```
__array_interface__
 Array protocol: Python side

__array_priority__
 Array priority.

__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

---

```
class object_(generic)
 Any Python object. Character code: '0'.
```

Method resolution order:

```
object_
generic
builtin object
```

---

Methods defined here:

```
add_(...)
 x. add_(y) <==> x+y

call_(...)
 x. call_(...) <==> x(...)

contains_(...)
 x. contains_(y) <==> y in x

delattr_(...)
 x. delattr_('name') <==> del x.name

delitem_(...)
```

```

x.__delitem__(y) <==> del x[y]

__eq__(...)
x.__eq__(y) <==> x==y

__ge__(...)
x.__ge__(y) <==> x>=y

__getattribute__(...)
x.__getattribute__('name') <==> x.name

__getitem__(...)
x.__getitem__(y) <==> x[y]

__gt__(...)
x.__gt__(y) <==> x>y

__hash__(...)
x.__hash__() <==> hash(x)

__iadd__(...)
x.__iadd__(y) <==> x+=y

__imul__(...)
x.__imul__(y) <==> x*=y

__le__(...)
x.__le__(y) <==> x<=y

__len__(...)
x.__len__() <==> len(x)

__lt__(...)
x.__lt__(y) <==> x<y

__mul__(...)
x.__mul__(n) <==> x*n

__ne__(...)
x.__ne__(y) <==> x!=y

__rmul__(...)
x.__rmul__(n) <==> n*x

__setattr__(...)
x.__setattr__('name', value) <==> x.name = value

__setitem__(...)
x.__setitem__(i, y) <==> x[i]=y

```

---

Data and other attributes defined here:

\_\_new\_\_ = <built-in method \_\_new\_\_ of type object>  
T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from generic:

```

__abs__(...)
x.__abs__() <==> abs(x)

__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

```

```
div(...)
x._div_(y) <==> x/y

divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

repr(...)
x._repr_() <==> repr(x)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x
```

**\_\_rmod\_\_(...)**  
x. rmod (y) <==> y%x

**\_\_ror\_\_(...)**  
x. ror (y) <==> y|x

**\_\_rpow\_\_(...)**  
y. rpow (x[, z]) <==> pow(x, y[, z])

**\_\_rrshift\_\_(...)**  
x. rrshift (y) <==> y>>x

**\_\_rshift\_\_(...)**  
x. rshift (y) <==> x>>y

**\_\_rsub\_\_(...)**  
x. rsub (y) <==> y-x

**\_\_rtruediv\_\_(...)**  
x. rtruediv (y) <==> y/x

**\_\_rxor\_\_(...)**  
x. rxor (y) <==> y^x

**\_\_setstate\_\_(...)**

**\_\_sizeof\_\_(...)**

**\_\_str\_\_(...)**  
x. str () <==> str(x)

**\_\_sub\_\_(...)**  
x. sub (y) <==> x-y

**\_\_truediv\_\_(...)**  
x. truediv (y) <==> x/y

**\_\_xor\_\_(...)**  
x. xor (y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

```

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New 'dtype' object with the given change to the byte order.

nonzero\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

```

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

class poly1d(_builtin_.object)
A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on
polynomials so that said operations may take on their customary
form in code (see Examples).

Parameters

c_or_r : array_like
 The polynomial's coefficients, in decreasing powers, or if
 the value of the second parameter is True, the polynomial's
 roots (values where the polynomial evaluates to 0). For example,
 ``poly1d([1, 2, 3])`` returns an object that represents
 :math:`x^2 + 2x + 3`, whereas ``poly1d([1, 2, 3], True)`` returns
 one that represents :math:`(x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6`.
r : bool, optional
 If True, `c_or_r` specifies the polynomial's roots; the default
 is False.
variable : str, optional
 Changes the variable used when printing `p` from `x` to `variable`
 (see Examples).

Examples

Construct the polynomial :math:`x^2 + 2x + 3`:

>>> p = np.poly1d([1, 2, 3])
>>> print(np.poly1d(p))
 2
1 x + 2 x + 3

Evaluate the polynomial at :math:`x = 0.5`:

>>> p(0.5)
4.25

Find the roots:

>>> p.r
array([-1.+1.41421356j, -1.-1.41421356j])
>>> p(p.r)
array([-4.44089210e-16+0.j, -4.44089210e-16+0.j])

These numbers in the previous line represent (0, 0) to machine precision

Show the coefficients:

>>> p.c
array([1, 2, 3])

Display the order (the leading zero-coefficients are removed):

>>> p.order
2

Show the coefficient of the k-th power in the polynomial
(which is equivalent to ``p.c[-(i+1)]``):

>>> p[1]
2

Polynomials can be added, subtracted, multiplied, and divided
(returns quotient and remainder):

>>> p * p
poly1d([1, 4, 10, 12, 9])
>>> (p**3 + 4) / p
(poly1d([1., 4., 10., 12., 9.]), poly1d([4.]))
``asarray(p)`` gives the coefficient array, so polynomials can be

```

```
used in all functions that accept arrays:
>>> p**2 # square of polynomial
poly1d([1, 4, 10, 12, 9])
>>> np.square(p) # square of individual coefficients
array([1, 4, 9])
```

The variable used in the string representation of `p` can be modified, using the `variable` parameter:

```
>>> p = np.poly1d([1,2,3], variable='z')
>>> print(p)
2
1 z + 2 z + 3
```

Construct a polynomial from its roots:

```
>>> np.poly1d([1, 2], True)
poly1d([1, -3, 2])
```

This is the same polynomial as obtained by:

```
>>> np.poly1d([1, -1]) * np.poly1d([1, -2])
poly1d([1, -3, 2])
```

Methods defined here:

\_\_add\_\_(self, other)  
\_\_array\_\_(self, t=None)  
\_\_call\_\_(self, val)  
\_\_div\_\_(self, other)  
\_\_eq\_\_(self, other)  
\_\_getattribute\_\_(self, key)  
\_\_getitem\_\_(self, val)  
\_\_init\_\_(self, c\_or\_r, r=0, variable=None)  
\_\_iter\_\_(self)  
\_\_len\_\_(self)  
\_\_mul\_\_(self, other)  
\_\_ne\_\_(self, other)  
\_\_neg\_\_(self)  
\_\_pos\_\_(self)  
\_\_pow\_\_(self, val)  
\_\_radd\_\_(self, other)  
\_\_rdiv\_\_(self, other)  
\_\_repr\_\_(self)  
\_\_rmul\_\_(self, other)  
\_\_rsub\_\_(self, other)  
\_\_rtruediv\_\_ = \_\_rdiv\_\_(self, other)  
\_\_setattr\_\_(self, key, val)  
\_\_setitem\_\_(self, key, val)  
\_\_str\_\_(self)  
\_\_sub\_\_(self, other)  
\_\_truediv\_\_ = \_\_div\_\_(self, other)

```
deriv(self, m=1)
 Return a derivative of this polynomial.

 Refer to `polyder` for full documentation.

See Also

polyder : equivalent function

integ(self, m=1, k=0)
 Return an antiderivative (indefinite integral) of this polynomial.

 Refer to `polyint` for full documentation.

See Also

polyint : equivalent function
```

---

Data descriptors defined here:

<b><u>__dict__</u></b>	dictionary for instance variables (if defined)
<b><u>__weakref__</u></b>	list of weak references to the object (if defined)

---

Data and other attributes defined here:

<b><u>__hash__</u></b> = None
<b><u>coeffs</u></b> = None
<b><u>order</u></b> = None
<b><u>variable</u></b> = None

---

```
class recarray(ndarray)
 Construct an ndarray that allows field access using attributes.

 Arrays may have a data-types containing fields, analogous
 to columns in a spread sheet. An example is ``[(x, int), (y, float)]``,
 where each entry in the array is a pair of ``(int, float)``. Normally,
 these attributes are accessed using dictionary lookups such as ``arr['x']``
 and ``arr['y']``. Record arrays allow the fields to be accessed as members
 of the array, using ``arr.x`` and ``arr.y``.

Parameters

shape : tuple
 Shape of output array.
dtype : data-type, optional
 The desired data-type. By default, the data-type is determined
 from `formats`, `names`, `titles`, `aligned` and `byteorder`.
formats : list of data-types, optional
 A list containing the data-types for the different columns, e.g.
 ``['i4', 'f8', 'i4']``. `formats` does *not* support the new
 convention of using types directly, i.e. ``(int, float, int)``.
 Note that `formats` must be a list, not a tuple.
 Given that `formats` is somewhat limited, we recommend specifying
 `dtype` instead.
names : tuple of str, optional
 The name of each column, e.g. ``('x', 'y', 'z')``.
buf : buffer, optional
 By default, a new array is created of the given shape and data-type.
 If `buf` is specified and is an object exposing the buffer interface,
 the array will use the memory from the existing buffer. In this case,
 the `offset` and `strides` keywords are available.

Other Parameters

titles : tuple of str, optional
 Aliases for column names. For example, if `names` were
 ``('x', 'y', 'z')`` and `titles` is
 ``('x_coordinate', 'y_coordinate', 'z_coordinate')``, then
 ``arr['x']`` is equivalent to both ``arr.x`` and ``arr.x_coordinate``.
byteorder : {'<', '>', '='}, optional
 Byte-order for all fields.
aligned : bool, optional
 Align the fields in memory as the C-compiler would.
strides : tuple of ints, optional
```

Buffer (`buf`) is interpreted according to these strides (strides define how many bytes each array element, row, column, etc. occupy in memory).

`offset : int, optional`  
Start reading buffer (`buf`) from this offset onwards.

`order : {'C', 'F'}, optional`  
Row-major (C-style) or column-major (Fortran-style) order.

**Returns**

-----

`rec : recarray`  
Empty array of the given shape and type.

**See Also**

-----

`rec.fromrecords` : Construct a [record](#) array from data.  
`record` : fundamental data-type for [recarray](#).  
`format parser` : determine a data-type from formats, names, titles.

**Notes**

-----

This constructor can be compared to ``empty``: it creates a new [record](#) array but does not fill it with data. To create a [record](#) array from data, use one of the following methods:

1. Create a standard [ndarray](#) and convert it to a [record](#) array, using ``arr.view(np.recarray)``
2. Use the `buf` keyword.
3. Use `np.rec.fromrecords`.

**Examples**

-----

Create an array with two fields, ``x`` and ``y``:

```
>>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', float), ('y', int)])
>>> x
array([(1.0, 2), (3.0, 4)],
 dtype=[('x', '<f8'), ('y', '<i4')])
```

```
>>> x['x']
array([1., 3.])
```

View the array as a [record](#) array:

```
>>> x = x.view(np.recarray)
>>> x.x
array([1., 3.])
```

```
>>> x.y
array([2, 4])
```

Create a new, empty [record](#) array:

```
>>> np.recarray((2,), ...
 ... dtype= [('x', int), ('y', float), ('z', int)]) #doctest: +SKIP
rec.array([-1073741821, 1.2249118382103472e-301, 24547520],
 (3471280, 1.2134086255804012e-316, 0)],
 dtype=[('x', '<i4'), ('y', '<f8'), ('z', '<i4')])
```

**Method resolution order:**

```
recarray
ndarray
builtin.object
```

---

Methods defined here:

```
__array_finalize__\(self, obj\)
__getattribute__\(self, attr\)
__getitem__\(self, indx\)
__repr__\(self\)
__setattr__\(self, attr, val\)
 # Save the dictionary.
 # If the attr is a field name and not in the saved dictionary
 # Undo any "setting" of the attribute and do a setfield
 # Thus, you can't create attributes on-the-fly that are field names.
field\(self, attr, val=None\)
```

---

Static methods defined here:

`_new_(subtype, shape, dtype=None, buf=None, offset=0, strides=None, formats=None, names=None, titles=None, byteorder=None, aligned=False, order='C')`

---

Data descriptors defined here:

**`_dict_`**

dictionary for instance variables (if defined)

---

Methods inherited from [ndarray](#):

**`_abs_()`**

`x._abs_() <=> abs(x)`

**`_add_()`**

`x._add_(y) <=> x+y`

**`_and_()`**

`x._and_(y) <=> x&y`

**`_array_()`**

`a._array_(|dtype) -> reference if type unchanged, copy otherwise.`

Returns either a new reference to self if `dtype` is not given or a new array of provided data type if `dtype` is different from the current `dtype` of the array.

**`_array_prepare_()`**

`a._array_prepare_(obj) -> Object of same type as ndarray object obj.`

**`_array_wrap_()`**

`a._array_wrap_(obj) -> Object of same type as ndarray object a.`

**`_contains_()`**

`x._contains_(y) <=> y in x`

**`_copy_()`**

`a._copy_([order])`

Return a copy of the array.

Parameters

-----

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

If order is 'C' (False) then the result is contiguous (default).

If order is 'Fortran' (True) then the result has fortran order.

If order is 'Any' (None) then the result has fortran order

only if the array already is in fortran order.

**`_deepcopy_()`**

`a._deepcopy_() -> Deep copy of array.`

Used if copy.deepcopy is called on an array.

**`_delitem_()`**

`x._delitem_(y) <=> del x[y]`

**`_delslice_()`**

`x._delslice_(i, j) <=> del x[i:j]`

Use of negative indices is not supported.

**`_div_()`**

`x._div_(y) <=> x/y`

**`_divmod_()`**

`x._divmod_(y) <=> divmod(x, y)`

**`_eq_()`**

`x._eq_(y) <=> x==y`

**`_float_()`**

`x._float_() <=> float(x)`

**`_floordiv_()`**

```
x. floordiv (y) <==> x//y
ge (...)
x. ge (y) <==> x>=y
getslice (...)
x. getslice (i, j) <==> x[i:j]
Use of negative indices is not supported.
gt (...)
x. gt (y) <==> x>y
hex (...)
x. hex () <==> hex(x)
iadd (...)
x. iadd (y) <==> x+=y
iand (...)
x. iand (y) <==> x&=y
idiv (...)
x. idiv (y) <==> x/=y
ifloordiv (...)
x. ifloordiv (y) <==> x//=y
ilshift (...)
x. ilshift (y) <==> x<=>y
imod (...)
x. imod (y) <==> x%>=y
imul (...)
x. imul (y) <==> x*>=y
index (...)
x[y:z] <==> x[y. index ():z. index ()]
int (...)
x. int () <==> int(x)
invert (...)
x. invert () <==> ~x
ior (...)
x. ior (y) <==> x|>=y
ipow (...)
x. ipow (y) <==> x**>=y
irshift (...)
x. irshift (y) <==> x>>=y
isub (...)
x. isub (y) <==> x->=y
iter (...)
x. iter () <==> iter(x)
itruediv (...)
x. itruediv (y) <==> x/>=y
ixor (...)
x. ixor (y) <==> x^>=y
le (...)
x. le (y) <==> x<=y
len (...)
x. len () <==> len(x)
long (...)
```

```
x. __long__() <==> long(x)

__lshift__(...)
x. __lshift__(y) <==> x<<y

__lt__(...)
x. __lt__(y) <==> x<y

__mod__(...)
x. __mod__(y) <==> x%y

__mul__(...)
x. __mul__(y) <==> x*y

__ne__(...)
x. __ne__(y) <==> x!=y

__neg__(...)
x. __neg__() <==> -x

__nonzero__(...)
x. __nonzero__() <==> x != 0

__oct__(...)
x. __oct__() <==> oct(x)

__or__(...)
x. __or__(y) <==> x|y

__pos__(...)
x. __pos__() <==> +x

__pow__(...)
x. __pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x. __radd__(y) <==> y+x

__rand__(...)
x. __rand__(y) <==> y&x

__rdiv__(...)
x. __rdiv__(y) <==> y/x

__rdivmod__(...)
x. __rdivmod__(y) <==> divmod(y, x)

__reduce__(...)
a. __reduce__()

 For pickling.

__rfloordiv__(...)
x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
x. __rlshift__(y) <==> y<<x

__rmod__(...)
x. __rmod__(y) <==> y%x

__rmul__(...)
x. __rmul__(y) <==> y*x

__ror__(...)
x. __ror__(y) <==> y|x

__rpow__(...)
y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x. __rrshift__(y) <==> y>>x

__rshift__(...)
```

```
x. rshift_(y) <==> x>>y
rsub_(...)
x. rsub_(y) <==> y-x

rtruediv_(...)
x. rtruediv_(y) <==> y/x

rxor_(...)
x. rxor_(y) <==> y^x

setitem_(...)
x. setitem_(i, y) <==> x[i]=y

setslice_(...)
x. setslice_(i, j, y) <==> x[i:j]=y

 Use of negative indices is not supported.

setstate_(...)
a. setstate_(version, shape, dtype, isfortran, rawdata)

 For unpickling.

Parameters

version : int
 optional pickle version. If omitted defaults to 0.
shape : tuple
dtype : data-type
isFortran : bool
rawdata : string or list
 a binary string with the data (or a list if 'a' is an object array)

sizeof_(...)

str_(...)
x. str_()
<==> str(x)

sub_(...)
x. sub_(y) <==> x-y

truediv_(...)
x. truediv_(y) <==> x/y

xor_(...)
x. xor_(y) <==> x^y

all(...)
a.all(axis=None, out=None, keepdims=False)

 Returns True if all elements evaluate to True.

 Refer to `numpy.all` for full documentation.

 See Also

numpy.all : equivalent function

any(...)
a.any(axis=None, out=None, keepdims=False)

 Returns True if any of the elements of `a` evaluate to True.

 Refer to `numpy.any` for full documentation.

 See Also

numpy.any : equivalent function

argmax(...)
a.argmax(axis=None, out=None)

 Return indices of the maximum values along the given axis.

 Refer to `numpy.argmax` for full documentation.

 See Also

numpy.argmax : equivalent function
```

```
argmin(...)
a.argmin(axis=None, out=None)

 Return indices of the minimum values along the given axis of `a`.

 Refer to `numpy.argmin` for detailed documentation.

See Also

numpy.argmax : equivalent function

argpartition(...)
a.argpartition(kth, axis=-1, kind='introselect', order=None)

 Returns the indices that would partition this array.

 Refer to `numpy.argpartition` for full documentation.

.. versionadded:: 1.8.0

See Also

numpy.argpartition : equivalent function

argsort(...)
a.argsort(axis=-1, kind='quicksort', order=None)

 Returns the indices that would sort this array.

 Refer to `numpy.argsort` for full documentation.

See Also

numpy.argsort : equivalent function

astype(...)
a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)

 Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype
 Typecode or data-type to which the array is cast.
order : {'C', 'F', 'A', 'K'}, optional
 Controls the memory layout order of the result.
 'C' means C order, 'F' means Fortran order, 'A'
 means 'F' order if all the arrays are Fortran contiguous,
 'C' order otherwise, and 'K' means as close to the
 order the array elements appear in memory as possible.
 Default is 'K'.
casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
 Controls what kind of data casting may occur. Defaults to 'unsafe'
 for backwards compatibility.

 * 'no' means the data types should not be cast at all.
 * 'equiv' means only byte-order changes are allowed.
 * 'safe' means only casts which can preserve values are allowed.
 * 'same_kind' means only safe casts or casts within a kind,
 like float64 to float32, are allowed.
 * 'unsafe' means any data conversions may be done.
subok : bool, optional
 If True, then sub-classes will be passed-through (default), otherwise
 the returned array will be forced to be a base-class array.
copy : bool, optional
 By default, astype always returns a newly allocated array. If this
 is set to false, and the `dtype`, `order`, and `subok`
 requirements are satisfied, the input array is returned instead
 of a copy.

Returns

arr_t : ndarray
 Unless `copy` is False and the other conditions for returning the input
 array are satisfied (see description for `copy` input parameter), `arr_t`
 is a new array of the same shape as the input array, with dtype, order
 given by dtype, order .

Notes

Starting in NumPy 1.9, astype method now returns an error if the string
dtype to cast to is not long enough in 'safe' casting mode to hold the max
value of integer/float array that is being casted. Previously the casting
was allowed even if the result was truncated.

Raises

```

```
ComplexWarning
 When casting from complex to float or int. To avoid this,
 one should use ``a.real.astype\(t\)``.

Examples

>>> x = np.array([1, 2, 2.5])
>>> x
array([1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])

byteswap(...)
a.byteswap(inplace)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by
returning a byteswapped array, optionally swapped in-place.

Parameters

inplace : bool, optional
 If ``True``, swap bytes in-place, default is ``False``.

Returns

out : ndarray
 The byteswapped array. If `inplace` is ``True``, this is
 a view to self.

Examples

>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
 dtype='|S3')

choose(...)
a.choose(choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also

numpy.choose : equivalent function

clip(...)
a.clip(min=None, max=None, out=None)

Return an array whose values are limited to ``[min, max]``.
One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See Also

numpy.clip : equivalent function

compress(...)
a.compress(condition, axis=None, out=None)

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

numpy.compress : equivalent function

conj(...)
a.conj()
```

```
Complex-conjugate all elements.
Refer to `numpy.conjugate` for full documentation.

See Also

numpy.conjugate : equivalent function

conjugate(...)
a.conjugate\(\)

Return the complex conjugate, element-wise.
Refer to `numpy.conjugate` for full documentation.

See Also

numpy.conjugate : equivalent function

copy(...)
a.copy\(order='C'\)

Return a copy of the array.

Parameters

order : {'C', 'F', 'A', 'K'}, optional
Controls the memory layout of the copy. 'C' means C-order,
'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
'C' otherwise. 'K' means match the layout of `a` as closely
as possible. (Note that this function and :func:`numpy.copy` are very
similar, but have different default values for their order=
arguments.)

See also

numpy.copy
numpy.copyto

Examples

>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy\(\)
>>> x.fill(0)

>>> x
array([[0, 0, 0],
 [0, 0, 0]])

>>> y
array([[1, 2, 3],
 [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True

cumprod(...)
a.cumprod\(axis=None, dtype=None, out=None\)

Return the cumulative product of the elements along the given axis.
Refer to `numpy.cumprod` for full documentation.

See Also

numpy.cumprod : equivalent function

cumsum(...)
a.cumsum\(axis=None, dtype=None, out=None\)

Return the cumulative sum of the elements along the given axis.
Refer to `numpy.cumsum` for full documentation.

See Also

numpy.cumsum : equivalent function

diagonal(...)
a.diagonal\(offset=0, axis1=0, axis2=1\)

Return specified diagonals. In NumPy 1.9 the returned array is a
read-only view instead of a copy as in previous NumPy versions. In
a future version the read-only restriction will be removed.
```

```
Refer to :func:`numpy.diagonal` for full documentation.

See Also

numpy.diagonal : equivalent function

dot(...)
a.dot(b, out=None)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See Also

numpy.dot : equivalent function

Examples

>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2., 2.],
 [2., 2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8., 8.],
 [8., 8.]])
```

**dump(...)**

```
a.dump(file)
```

Dump a pickle of the array to the specified file.  
The array can be read back with pickle.load or numpy.load.

Parameters

```

```

file : [str](#)  
A string naming the dump file.

**dumps(...)**

```
a.dumps()
```

Returns the pickle of the array as a string.  
pickle.loads or numpy.loads will convert the string back to an array.

Parameters

```

```

None

**fill(...)**

```
a.fill(value)
```

Fill the array with a scalar value.

Parameters

```

```

value : scalar  
All elements of `a` will be assigned this value.

Examples

```

```

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```

**flatten(...)**

```
a.flatten(order='C')
```

Return a copy of the array collapsed into one dimension.

Parameters

```

```

order : {'C', 'F', 'A', 'K'}, optional  
'C' means to flatten in row-major (C-style) order.  
'F' means to flatten in column-major (Fortran-style) order.  
'A' means to flatten in column-major order if `a` is Fortran \*contiguous\* in memory,

```
row-major order otherwise. 'K' means to flatten
`a` in the order the elements occur in memory.
The default is 'C'.

>Returns

y : ndarray
 A copy of the input array, flattened to one dimension.

See Also

ravel : Return a flattened array.
flat : A 1-D flat iterator over the array.

Examples

>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])

getfield(...)
a.getfield(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in
the view are determined by the given type and the offset into the current
array in bytes. The offset needs to be such that the view dtype fits in the
array dtype; for example an array of dtype complex128 has 16-byte elements.
If taking a view with a 32-bit integer (4 bytes), the offset needs to be
between 0 and 12 bytes.

Parameters

dtype : str or dtype
 The data type of the view. The dtype size of the view can not be larger
 than that of the array itself.
offset : int
 Number of bytes to skip before beginning the element view.

Examples

>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j, 0.+0.j],
 [0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[1., 0.],
 [0., 2.]))

By choosing an offset of 8 bytes we can select the complex part of the
array for our view:

>>> x.getfield(np.float64, offset=8)
array([[1., 0.],
 [0., 4.]])

item(...)
a.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

* none: in this case, the method only works for arrays
 with one element (`a.size == 1`), which element is
 copied into a standard Python scalar object and returned.

* int_type: this argument is interpreted as a flat index into
 the array, specifying which element to copy and return.

* tuple of int_types: functions as does a single int_type argument,
 except that the argument is interpreted as an nd-index into the
 array.

>Returns

z : Standard Python scalar object
 A copy of the specified element of the array as a suitable
 Python scalar

Notes

```

When the data type of `a` is `longdouble` or `clongdouble`, `item()` returns a scalar array `object` because there is no available Python scalar that would not lose information. Void arrays return a buffer `object` for `item()`, unless fields are defined, in which case a tuple is returned.

`item` is very similar to `a[args]`, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

**Examples**

```

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

**itemset(...)**

```
a.itemset(*args)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument
as *item*. Then, ``a.itemset(*args)`` is equivalent to but faster
than ``a[args] = item``. The item should be a scalar value and `args`
must select a single item in the array `a`.
```

**Parameters**

```

*args : Arguments
 If one argument: a scalar, only used in case `a` is of size 1.
 If two arguments: the last argument is the value to be set
 and must be a scalar, the first argument specifies a single array
 element location. It is either an int or a tuple.
```

**Notes**

```

Compared to indexing syntax, `itemset` provides some speed increase
for placing a scalar into a particular location in an ndarray,
if you must do this. However, generally this is discouraged:
among other problems, it complicates the appearance of the code.
Also, when using `itemset` (and `item`) inside a loop, be sure
to assign the methods to a local variable to avoid the attribute
look-up at each loop iteration.
```

**Examples**

```

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
 [2, 8, 3],
 [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
 [2, 0, 3],
 [8, 5, 9]])
```

**max(...)**

```
a.max(axis=None, out=None)

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.
```

**See Also**

```

numpy.amax : equivalent function
```

**mean(...)**

```
a.mean(axis=None, dtype=None, out=None, keepdims=False)

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.
```

**See Also**

```

```

```
 numpy.mean : equivalent function

min(...)
 a.min(axis=None, out=None, keepdims=False)

 Return the minimum along a given axis.

 Refer to `numpy.amin` for full documentation.

 See Also

 numpy.amin : equivalent function

newbyteorder(...)
 arr.newbyteorder(new_order='S')

 Return the array with the same data viewed with a different byte order.

 Equivalent to::

 arr.view(arr.dtype.newbyteorder(new_order))

 Changes are also made in all fields and sub-arrays of the array data
 type.

 Parameters

 new_order : string, optional
 Byte order to force; a value from the byte order specifications
 below. `new_order` codes can be any of:
 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_arr : array
 New array object with the dtype reflecting given change to the
 byte order.

nonzero(...)
 a.nonzero()

 Return the indices of the elements that are non-zero.

 Refer to `numpy.nonzero` for full documentation.

 See Also

 numpy.nonzero : equivalent function

partition(...)
 a.partition(kth, axis=-1, kind='introselect', order=None)

 Rearranges the elements in the array in such a way that value of the
 element in kth position is in the position it would be in a sorted array.
 All elements smaller than the kth element are moved before this element and
 all equal or greater are moved behind it. The ordering of the elements in
 the two partitions is undefined.

 .. versionadded:: 1.8.0

 Parameters

 kth : int or sequence of ints
 Element index to partition by. The kth element value will be in its
 final sorted position and all smaller elements will be moved before it
 and all equal or greater elements behind it.
 The order all elements in the partitions is undefined.
 If provided with a sequence of kth it will partition all elements
 indexed by kth of them into their sorted position at once.
 axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
 kind : {'introselect'}, optional
 Selection algorithm. Default is 'introselect'.
```

```
order : str or list of str, optional
 When 'a' is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.partition : Return a partitioned copy of an array.
argpartition : Indirect partition.
sort : Full sort.

Notes

See ```np.partition`` for notes on the different algorithms.

Examples

>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])

prod(...)
a.prod(axis=None, dtype=None, out=None, keepdims=False)

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also

numpy.prod : equivalent function

ptp(...)
a.ptp(axis=None, out=None)

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also

numpy.ptp : equivalent function

put(...)
a.put(indices, values, mode='raise')

Set ```a.flat[n] = values[n]`` for all `n` in indices.

Refer to `numpy.put` for full documentation.

See Also

numpy.put : equivalent function

ravel(...)
a.ravel([order])

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See Also

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat(...)
a.repeat(repeats, axis=None)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also

numpy.repeat : equivalent function

reshape(...)
a.reshape(shape, order='C')
```

```
Returns an array containing the same data with a new shape.
Refer to `numpy.reshape` for full documentation.
See Also

numpy.reshape : equivalent function
resize(...)
a.resize(new_shape, refcheck=True)
Change shape and size of array in-place.
Parameters

new_shape : tuple of ints, or `n` ints
 Shape of resized array.
refcheck : bool, optional
 If False, reference count will not be checked. Default is True.
Returns

None
Raises

ValueError
 If `a` does not own its own data or references or views to it exist,
 and the data memory must be changed.
SystemError
 If the `order` keyword argument is specified. This behaviour is a
 bug in NumPy.
See Also

resize : Return a new array with the specified shape.
Notes

This reallocates space for the data area if necessary.
Only contiguous arrays (data elements consecutive in memory) can be
resized.
The purpose of the reference count check is to make sure you
do not use this array as a buffer for another Python object and then
reallocate the memory. However, reference counts can increase in
other ways so if you are sure that you have not shared the memory
for this array with another Python object, then you may safely set
`refcheck` to False.
Examples

Shrinking an array: array is flattened (in the order that the data are
stored in memory), resized, and reshaped:

>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
 [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
 [2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
 [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
Unless `refcheck` is False:

>>> a.resize((1, 1), refcheck=False)
```

```
>>> a
array([[0]])
>>> c
array([[0]])

round(...)
a.round(decimals=0, out=None)

 Return `a` with each element rounded to the given number of decimals.

 Refer to `numpy.around` for full documentation.

See Also

numpy.around : equivalent function

searchsorted(...)
a.searchsorted(v, side='left', sorter=None)

 Find indices where elements of v should be inserted in a to maintain order.

 For full documentation, see `numpy.searchsorted`

See Also

numpy.searchsorted : equivalent function

setfield(...)
a.setfield(val, dtype, offset=0)

 Put a value into a specified place in a field defined by a data-type.

 Place `val` into `a`'s field defined by `dtype` and beginning `offset` bytes into the field.

Parameters

val : object
 Value to be placed in field.
dtype : dtype object
 Data-type of the field in which to place `val`.
offset : int, optional
 The number of bytes into the field at which to place `val`.

Returns

None

See Also

getfield

Examples

>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
 [3, 3, 3],
 [3, 3, 3]])
>>> x
array([[1.0000000e+000, 1.48219694e-323, 1.48219694e-323],
 [1.48219694e-323, 1.0000000e+000, 1.48219694e-323],
 [1.48219694e-323, 1.48219694e-323, 1.0000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
```

  

```
setflags(...)
a.setflags(write=None, align=None, uic=None)

 Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

 These Boolean-valued flags affect how numpy interprets the memory area used by `a` (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)
```

```

Parameters

write : bool, optional
 Describes whether or not `a` can be written to.
align : bool, optional
 Describes whether or not `a` is aligned properly for its type.
uic : bool, optional
 Describes whether or not `a` is a copy of another "base" array.

Notes

Array flags provide information about how the memory area used
for the array is to be interpreted. There are 6 Boolean flags
in use, only three of which can be changed by the user:
UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware
(as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced
by .base). When this array is deallocated, the base array will be
updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well
as the full name.

Examples

>>> y
array([[3, 1, 7],
 [2, 0, 0],
 [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True

sort(...)
a.sort(axis=-1, kind='quicksort', order=None)

Sort an array, in-place.

Parameters

axis : int, optional
 Axis along which to sort. Default is -1, which means sort along the
 last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 Sorting algorithm. Default is 'quicksort'.
order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

See Also

numpy.sort : Return a sorted copy of an array.
argsort : Indirect sort.
lexsort : Indirect stable sort on multiple keys.
searchsorted : Find elements in sorted array.
partition: Partial sort.

Notes

See ``sort`` for notes on the different sorting algorithms.

Examples

>>> a = np.array([[1,4], [3,1]])

```

```
>>> a.sort(axis=1)
>>> a
array([[1, 4],
 [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
 [1, 4]])

Use the `order` keyword to specify a field to use when sorting a
structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
 dtype=[('x', '|S1'), ('y', '<i4')])

squeeze(...)
a.squeeze(axis=None)

Remove single-dimensional entries from the shape of `a`.

Refer to `numpy.squeeze` for full documentation.

See Also

numpy.squeeze : equivalent function

std(...)
a.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also

numpy.std : equivalent function

sum(...)
a.sum(axis=None, dtype=None, out=None, keepdims=False)

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also

numpy.sum : equivalent function

swapaxes(...)
a.swapaxes(axis1, axis2)

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

numpy.swapaxes : equivalent function

take(...)
a.take(indices, axis=None, out=None, mode='raise')

Return an array formed from the elements of `a` at the given indices.

Refer to `numpy.take` for full documentation.

See Also

numpy.take : equivalent function

tobytes(...)
a.tobytes(order='C')

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of
data memory. The bytes object can be produced in either 'C' or 'Fortran',
or 'Any' order (the default is 'C'-order). 'Any' order means C-order
unless the F_CONTIGUOUS flag in the array is set, in which case it
means 'Fortran' order.

.. versionadded:: 1.9.0
```

**Parameters**  
-----  
`order : {'C', 'F', None}, optional`  
    Order of the data for multidimensional arrays:  
    C, Fortran, or the same as for the original array.

**Returns**  
-----  
`s : bytes`  
    Python bytes exhibiting a copy of `a`'s raw data.

**Examples**  
-----  

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x00\x03\x00\x00\x00'
```

**tofile(...)**  
`a.tofile(fid, sep="", format="%s")`  
Write array to a file as text or binary (default).  
Data is always written in 'C' order, independent of the order of `a`. The data produced by this method can be recovered using the function `fromfile()`.

**Parameters**  
-----  
`fid : file or str`  
    An open file object, or a string containing a filename.  
`sep : str`  
    Separator between array items for text output.  
    If "" (empty), a binary file is written, equivalent to  
    ``file.write(a.tobytes())``.  
`format : str`  
    Format string for text file output.  
    Each entry in the array is formatted to text by first converting  
    it to the closest Python type, and then using "format" % item.

**Notes**  
-----  
This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

**tolist(...)**  
`a.tolist()`  
Return the array as a (possibly nested) list.  
Return a copy of the array data as a (nested) Python list.  
Data items are converted to the nearest compatible Python type.

**Parameters**  
-----  
`none`

**Returns**  
-----  
`y : list`  
    The possibly nested list of array elements.

**Notes**  
-----  
The array may be recreated, ``a = np.array(a.tolist())``.

**Examples**  
-----  

```
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

**tosstring(...)**  
`a.tosstring(order='C')`  
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes [object](#) can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the `F_CONTIGUOUS` flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for `tobytes`. Despite its name it returns bytes not strings.

**Parameters**

`order : {'C', 'F', None}, optional`  
Order of the data for multidimensional arrays:  
C, Fortran, or the same as for the original array.

**Returns**

`s : bytes`  
Python bytes exhibiting a copy of `a`'s raw data.

**Examples**

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

**trace(...)**  
`a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

**See Also**

`numpy.trace` : equivalent function

**transpose(...)**  
`a.transpose(*axes)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a [matrix object](#).)  
For a 2-D array, this is the usual [matrix](#) transpose.  
For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

`axes : None, tuple of ints, or 'n' ints`

- \* None or no argument: reverses the order of the axes.
- \* tuple of ints: `i` in the `j`-th place in the tuple means `a`'s `i`-th axis becomes `a.transpose()`'s `j`-th axis.
- \* `n` ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

`out : ndarray`  
View of `a`, with axes suitably permuted.

**See Also**

`ndarray.T` : Array property returning the array transposed.

**Examples**

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
 [3, 4]])
>>> a.transpose()
array([[1, 3],
 [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
 [2, 4]])
>>> a.transpose(1, 0)
```

```
array([[1, 3],
 [2, 4]])

var(...)
 a.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

 Returns the variance of the array elements, along given axis.

 Refer to `numpy.var` for full documentation.

 See Also

 numpy.var : equivalent function

view(...)
 a.view(dtype=None, type=None)

 New view of array with the same data.

 Parameters

 dtype : data-type or ndarray sub-class, optional
 Data-type descriptor of the returned view, e.g., float32 or int16. The
 default, None, results in the view having the same data-type as 'a'.
 This argument can also be specified as an ndarray sub-class, which
 then specifies the type of the returned object (this is equivalent to
 setting the ``type`` parameter).
 type : Python type, optional
 Type of the returned view, e.g., ndarray or matrix. Again, the
 default None results in type preservation.

 Notes

 ``a.view()`` is used two different ways:

 ``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a view
 of the array's memory with a different data-type. This can cause a
 reinterpretation of the bytes of memory.

 ``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` just
 returns an instance of `ndarray_subclass` that looks at the same array
 (same shape, dtype, etc.) This does not cause a reinterpretation of the
 memory.

 For ``a.view(some_dtype)``, if ``some_dtype`` has a different number of
 bytes per entry than the previous dtype (for example, converting a
 regular array to a structured array), then the behavior of the view
 cannot be predicted just from the superficial appearance of ``a`` (shown
 by ``print(a)``). It also depends on exactly how ``a`` is stored in
 memory. Therefore if ``a`` is C-ordered versus fortran-ordered, versus
 defined as a slice or transpose, etc., the view may give different
 results.

 Examples

 >>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
 Viewing array data using a different type and dtype:
 >>> y = x.view(dtype=np.int16, type=np.matrix)
 >>> y
 matrix([[513]], dtype=int16)
 >>> print(type(y))
 <class 'numpy.matrixlib.defmatrix.matrix'>
 Creating a view on a structured array so it can be used in calculations
 >>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
 >>> xv = x.view(dtype=np.int8).reshape(-1,2)
 >>> xv
 array([[1, 2],
 [3, 4]], dtype=int8)
 >>> xv.mean(0)
 array([2., 3.])

 Making changes to the view changes the underlying array
 >>> xv[0,1] = 20
 >>> print(x)
 [(1, 20) (3, 4)]

 Using a view to convert an array to a recarray:
 >>> z = x.view(np.recarray)
 >>> z.a
 array([1], dtype=int8)
```

```

Views share data:

>>> x[0] = (9, 10)
>>> z[0]
(9, 10)

Views that change the dtype size (bytes per entry) should normally be
avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
 [4, 5]], dtype=int16)
>>> y.view(dtype=\[\('width', np.int16\), \('length', np.int16\)\])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=\[\('width', np.int16\), \('length', np.int16\)\])
array([[1, 2],
 [4, 5]], dtype=\[\('width', '<i2'\), \('length', '<i2'\)\])

```

---

Data descriptors inherited from [ndarray](#):

## T

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

### Examples

```

>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[1., 2.],
 [3., 4.]])
>>> x.T
array([[1., 3.],
 [2., 4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([1., 2., 3., 4.])
>>> x.T
array([1., 2., 3., 4.])

```

## [\\_\\_array\\_interface\\_\\_](#)

Array protocol: Python side.

## [\\_\\_array\\_priority\\_\\_](#)

Array priority.

## [\\_\\_array\\_struct\\_\\_](#)

Array protocol: C-struct side.

## [base](#)

Base object if memory is from some other object.

### Examples

The base of an array that owns its memory is `None`:

```

>>> x = np.array([1,2,3,4])
>>> x.base is None
True

```

Slicing creates a view, whose memory is shared with `x`:

```

>>> y = x[2:]
>>> y.base is x
True

```

## [ctypes](#)

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

### Parameters

-----

`None`

### Returns

```

c : Python object
 Possessing attributes data, shape, strides, etc.

See Also

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented
in "Guide to NumPy" (we have omitted undocumented public attributes,
as well as documented private attributes):

* data: A pointer to the memory area of the array as a Python integer.
 This memory area may contain data that is not aligned, or not in correct
 byte-order. The memory area may not even be writeable. The array
 flags and data-type of this array should be respected when passing this
 attribute to arbitrary C-code to avoid trouble that can include Python
 crashing. User Beware! The value of this attribute is exactly the same
 as self._array_interface_['data'][0].

* shape (c_intp*self.ndim): A ctypes array of length self.ndim where
 the basetype is the C-integer corresponding to dtype('p') on this
 platform. This base-type could be c_int, c_long, or c_longlong
 depending on the platform. The c_intp type is defined accordingly in
 numpy.ctypeslib. The ctypes array contains the shape of the underlying
 array.

* strides (c_intp*self.ndim): A ctypes array of length self.ndim where
 the basetype is the same as for the shape attribute. This ctypes array
 contains the strides information from the underlying array. This strides
 information is important for showing how many bytes must be jumped to
 get to the next element in the array.

* data_as(obj): Return the data pointer cast to a particular c-types object.
 For example, calling self._as_parameter_ is equivalent to
 self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a
 pointer to a ctypes array of floating-point data:
 self.data_as(ctypes.POINTER(ctypes.c_double)).

* shape_as(obj): Return the shape tuple as an array of some other c-types
 type. For example: self.shape_as(ctypes.c_short).

* strides_as(obj): Return the strides tuple as an array of some other
 c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary
arrays or arrays constructed on the fly. For example, calling
``(a+b).ctypes.data_as(ctypes.c_void_p)`` returns a pointer to memory
that is invalid because the array created as (a+b) is deallocated
before the next Python statement. You can avoid this problem using
either ``c=a+b`` or ``ct=(a+b).ctypes``. In the latter case, ct will
hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute
of array objects still returns something useful, but ctypes objects
are not returned and errors may be raised instead. In particular,
the object will still have the as parameter attribute which will
return an integer equal to the data attribute.

Examples

>>> import ctypes
>>> x
array([[0, 1],
 [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

**data**

Python buffer object pointing to the start of the array's data.

**dtype**

```

Data-type of the array's elements.

Parameters

None

Returns

d : numpy dtype object

See Also

numpy.dtype

Examples

>>> x
array([[0, 1],
 [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>

```

**flags**  
Information about the memory layout of the array.

Attributes

- 
- C\_CONTIGUOUS (C)  
The data is in a single, C-style contiguous segment.
- F\_CONTIGUOUS (F)  
The data is in a single, Fortran-style contiguous segment.
- OWNDATA (O)  
The array owns the memory it uses or borrows it from another object.
- WRITEABLE (W)  
The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
- ALIGNED (A)  
The data and all elements are aligned appropriately for the hardware.
- UPDATEIFCOPY (U)  
This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
- FNC  
F\_CONTIGUOUS and not C\_CONTIGUOUS.
- FORC  
F\_CONTIGUOUS or C\_CONTIGUOUS (one-segment test).
- BEHAVED (B)  
ALIGNED and WRITEABLE.
- CARRAY (CA)  
BEHAVED and C\_CONTIGUOUS.
- FARRAY (FA)  
BEHAVED and F\_CONTIGUOUS and not C\_CONTIGUOUS.

Notes

- 
- The `flags` object can be accessed dictionary-like (as in ``a.flags['WRITEABLE']``), or by using lowercased attribute names (as in ``a.flags.writeable``). Short flag names are only supported in dictionary access.
- Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.
- The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set ``False``.
- ALIGNED can only be set ``True`` if the data is truly aligned.
- WRITEABLE can only be set ``True`` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension ``arr.strides[dim]`` may be \*arbitrary\* if ``arr.shape[dim] == 1`` or the array has no elements.  
It does \*not\* generally hold that ``self.strides[-1] == self.itemsize``

```
for C-style contiguous arrays or ``self.strides[0] == self.itemsize`` for
Fortran-style contiguous arrays is true.
```

**flat**

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

-----  
flatten : Return a copy of the array collapsed into one dimension.

**flatiter**

Examples

```
----->>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
 [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
 [2, 5],
 [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
 [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
 [3, 1, 3]])
```

**imag**

The imaginary part of the array.

Examples

```
----->>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

**itemsize**

Length of one array element in bytes.

Examples

```
----->>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

**nbytes**

Total bytes consumed by the elements of the array.

Notes

-----Does not include memory consumed by non-element attributes of the array object.

Examples

```
----->>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**ndim**

Number of array dimensions.

Examples

```
----->>> x = np.array([1, 2, 3])
```

```

>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3

real
The real part of the array.

Examples

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([1. , 0.70710678])
>>> x.real.dtype
dtype('float64')

See Also

numpy.real : equivalent function

```

**shape**  
Tuple of array dimensions.

**Notes**

-----

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

**Examples**

-----

```

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged

```

**size**  
Number of elements in the array.

Equivalent to ``np.prod(a.shape)``, i.e., the product of the array's dimensions.

**Examples**

-----

```

>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30

```

**strides**  
Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ``i[0], i[1], ..., i[n]`` in an array `a`  
is:::

```

offset = sum(np.array(i) * a.strides)

```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

**Notes**

-----

Imagine an array of 32-bit integers (each 4 bytes):::

```

x = np.array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]], dtype=np.int32)

```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be ``(20, 4)``.

```

See Also

numpy.lib.stride_tricks.as_strided

Examples

>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]],
 [[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813

```

---

Data and other attributes inherited from [ndarray](#):

**\_\_hash\_\_** = None

### class record([void](#))

A data-type scalar that allows field access as attribute lookup.

Method resolution order:

[record](#)  
[void](#)  
[flexible](#)  
[generic](#)  
[builtin\\_object](#)

---

Methods defined here:

**\_\_getattribute\_\_**(self, attr)  
**\_\_getitem\_\_**(self, indx)  
**\_\_repr\_\_**(self)  
**\_\_setattr\_\_**(self, attr, val)  
**\_\_str\_\_**(self)  
**pprint**(self)  
     Pretty-print all fields.

---

Data descriptors defined here:

**\_\_dict\_\_**  
     dictionary for instance variables (if defined)  
**\_\_weakref\_\_**  
     list of weak references to the object (if defined)

---

Methods inherited from [void](#):

**\_\_delitem\_\_**(...)  
     x.[\\_\\_delitem\\_\\_](#)(y) <=> del x[y]

---

**\_\_eq\_\_**(...)  
 x.\_\_eq\_\_(y) <==> x==y  
**\_\_ge\_\_**(...)  
 x.\_\_ge\_\_(y) <==> x>=y  
**\_\_gt\_\_**(...)  
 x.\_\_gt\_\_(y) <==> x>y  
**\_\_hash\_\_**(...)  
 x.\_\_hash\_\_() <==> hash(x)  
**\_\_le\_\_**(...)  
 x.\_\_le\_\_(y) <==> x<=y  
**\_\_len\_\_**(...)  
 x.\_\_len\_\_() <==> len(x)  
**\_\_lt\_\_**(...)  
 x.\_\_lt\_\_(y) <==> x<y  
**\_\_ne\_\_**(...)  
 x.\_\_ne\_\_(y) <==> x!=y  
**\_\_setitem\_\_**(...)  
 x.\_\_setitem\_\_(i, y) <==> x[i]=y  
**getfield**(...)  
**setfield**(...)

---

Data descriptors inherited from [void](#):

**dtype**  
 dtype object

**flags**  
 integer value of flags

---

Data and other attributes inherited from [void](#):

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
 T.\_\_new\_\_(S, ...) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [generic](#):

**\_\_abs\_\_**(...)  
 x.\_\_abs\_\_() <==> abs(x)  
**\_\_add\_\_**(...)  
 x.\_\_add\_\_(y) <==> x+y  
**\_\_and\_\_**(...)  
 x.\_\_and\_\_(y) <==> x&y  
**\_\_array\_\_**(...)  
 sc.\_\_array\_\_(|type) return 0-dim array  
**\_\_array\_wrap\_\_**(...)  
 sc.\_\_array\_wrap\_\_(obj) return scalar from array  
**\_\_copy\_\_**(...)  
**\_\_deepcopy\_\_**(...)  
**\_\_div\_\_**(...)  
 x.\_\_div\_\_(y) <==> x/y  
**\_\_divmod\_\_**(...)  
 x.\_\_divmod\_\_(y) <==> divmod(x, y)  
**\_\_float\_\_**(...)

```
x.float() <==> float(x)

floordiv(...)
x.floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

hex(...)
x.hex() <==> hex(x)

int(...)
x.int() <==> int(x)

invert(...)
x.invert() <==> ~x

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

neg(...)
x.neg() <==> -x

nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
```

```
x.__ror__(y) <==> y|x
```

\_\_rpow\_\_(...)  
y.rpow(x[, z]) <==> pow(x, y[, z])

\_\_rrshift\_\_(...)  
x.\_\_rrshift\_\_(y) <==> y>>x

\_\_rshift\_\_(...)  
x.\_\_rshift\_\_(y) <==> x>>y

\_\_rsub\_\_(...)  
x.\_\_rsub\_\_(y) <==> y-x

\_\_rtruediv\_\_(...)  
x.\_\_rtruediv\_\_(y) <==> y/x

\_\_rxor\_\_(...)  
x.\_\_rxor\_\_(y) <==> y^x

\_\_setstate\_\_(...)

\_\_sizeof\_\_(...)

\_\_sub\_\_(...)  
x.\_\_sub\_\_(y) <==> x-y

\_\_truediv\_\_(...)  
x.\_\_truediv\_\_(y) <==> x/y

\_\_xor\_\_(...)  
x.\_\_xor\_\_(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
 so as to provide a uniform API.
See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

**short** = class int16([signedinteger](#))  
16-bit [integer](#). Character code ``h''. C [short](#) compatible.

Method resolution order:

[int16](#)  
[signedinteger](#)  
[integer](#)  
[number](#)  
[generic](#)  
[\\_builtin\\_object](#)

---

Methods defined here:

**\_\_eq\_\_(...)**  
x.[\\_\\_eq\\_\\_](#)(y) <==> x==y

**\_\_ge\_\_(...)**  
x.[\\_\\_ge\\_\\_](#)(y) <==> x>=y

---

```
__gt__(...)
x.__gt__(y) <==> x>y

__hash__(...)
x.__hash__() <==> hash(x)

__index__(...)
x[y:z] <==> x[y.__index__():z.__index__()]

__le__(...)
x.__le__(y) <==> x<=y

__lt__(...)
x.__lt__(y) <==> x<y

__ne__(...)
x.__ne__(y) <==> x!=y
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

```
__abs__(...)
x.__abs__() <==> abs(x)

__add__(...)
x.__add__(y) <==> x+y

__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type|) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)
```

```
int(...)
x._int_() <==> int(x)
invert(...)
x._invert_() <==> ~x
long(...)
x._long_() <==> long(x)
lshift(...)
x._lshift_(y) <==> x<<y
mod(...)
x._mod_(y) <==> x%y
mul(...)
x._mul_(y) <==> x*y
neg(...)
x._neg_() <==> -x
nonzero(...)
x._nonzero_() <==> x != 0
oct(...)
x._oct_() <==> oct(x)
or(...)
x._or_(y) <==> x|y
pos(...)
x._pos_() <==> +x
pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])
radd(...)
x._radd_(y) <==> y+x
rand(...)
x._rand_(y) <==> y&x
rdiv(...)
x._rdiv_(y) <==> y/x
rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)
reduce(...)
repr(...)
x._repr_() <==> repr(x)
rfloordiv(...)
x._rfloordiv_(y) <==> y//x
rlshift(...)
x._rlshift_(y) <==> y<<x
rmod(...)
x._rmod_(y) <==> y%x
rmul(...)
x._rmul_(y) <==> y*x
ror(...)
x._ror_(y) <==> y|x
rpow(...)
y._rpow_(x[, z]) <==> pow(x, y[, z])
rrshift(...)
x._rrshift_(y) <==> y>>x
```

```
__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

```

The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new '[dtype](#)' with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----

new\_order : [str](#), optional  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----

new\_dtype : [dtype](#)  
New '[dtype](#)' [object](#) with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_array\_interface\_**  
Array protocol: Python side

**\_array\_priority\_**  
Array priority.

**\_array\_struct\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

```
class signedinteger(integer)
 Method resolution order:
 signedinteger
 integer
 number
 generic
 builtin__object
```

---

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**

x.[\\_\\_abs\\_\\_](#)() <==> abs(x)

**\_\_add\_\_(...)**

x.[\\_\\_add\\_\\_](#)(y) <==> x+y

**\_\_and\_\_(...)**

x.[\\_\\_and\\_\\_](#)(y) <==> x&y

**\_\_array\_\_(...)**

sc.[\\_\\_array\\_\\_](#)(|type|) return 0-dim array

**\_\_array\_wrap\_\_(...)**

sc.[\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**

x.[\\_\\_div\\_\\_](#)(y) <==> x/y

**\_\_divmod\_\_(...)**

x.[\\_\\_divmod\\_\\_](#)(y) <==> divmod(x, y)

**\_\_eq\_\_(...)**

x.[\\_\\_eq\\_\\_](#)(y) <==> x==y

**\_\_float\_\_(...)**

x.[\\_\\_float\\_\\_](#)() <==> float(x)

**\_\_floordiv\_\_(...)**

x.[\\_\\_floordiv\\_\\_](#)(y) <==> x//y

**\_\_format\_\_(...)**

NumPy array scalar formatter

**\_\_ge\_\_(...)**

x.[\\_\\_ge\\_\\_](#)(y) <==> x>=y

**\_\_getitem\_\_(...)**

x.[\\_\\_getitem\\_\\_](#)(y) <==> x[y]

**\_\_gt\_\_(...)**

x.[\\_\\_gt\\_\\_](#)(y) <==> x>y

**\_\_hex\_\_(...)**

x.[\\_\\_hex\\_\\_](#)() <==> hex(x)

**\_\_int\_\_(...)**

x.[\\_\\_int\\_\\_](#)() <==> int(x)

**\_\_invert\_\_(...)**

```
x. invert () <==> ~x
le (...)
x. le (y) <==> x<=y
long (...)
x. long () <==> long(x)
lshift (...)
x. lshift (y) <==> x<<y
lt (...)
x. lt (y) <==> x<y
mod (...)
x. mod (y) <==> x%y
mul (...)
x. mul (y) <==> x*y
ne (...)
x. ne (y) <==> x!=y
neg (...)
x. neg () <==> -x
nonzero (...)
x. nonzero () <==> x != 0
oct (...)
x. oct () <==> oct(x)
or (...)
x. or (y) <==> x|y
pos (...)
x. pos () <==> +x
pow (...)
x. pow (y[, z]) <==> pow(x, y[, z])
radd (...)
x. radd (y) <==> y+x
rand (...)
x. rand (y) <==> y&x
rdiv (...)
x. rdiv (y) <==> y/x
rdivmod (...)
x. rdivmod (y) <==> divmod(y, x)
reduce (...)
repr (...)
x. repr () <==> repr(x)
rfloordiv (...)
x. rfloordiv (y) <==> y//x
rlshift (...)
x. rlshift (y) <==> y<<x
rmod (...)
x. rmod (y) <==> y%x
rmul (...)
x. rmul (y) <==> y*x
ror (...)
x. ror (y) <==> y|x
rpow (...)
```

```
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
 x.rrshift(y) <==> y>>x

rshift(...)
 x.rshift(y) <==> x>>y

rsub(...)
 x.rsub(y) <==> y-x

rtruediv(...)
 x.rtruediv(y) <==> y/x

rxor(...)
 x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
 x.str() <==> str(x)

sub(...)
 x.sub(y) <==> x-y

truediv(...)
 x.truediv(y) <==> x/y

xor(...)
 x.xor(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.
```

```
Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

repeat(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

reshape(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

resize(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

---

**single** = class float32([floating](#))  
32-bit [floating-point number](#). Character code 'f'. C [float](#) compatible.

Method resolution order:

[float32](#)  
[floating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[\\_builtin\\_.object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_\(...\)](#)  
x. [\\_\\_eq\\_\\_\(y\)](#) <==> x==y

[\\_\\_ge\\_\\_\(...\)](#)  
x. [\\_\\_ge\\_\\_\(y\)](#) <==> x>=y

[\\_\\_gt\\_\\_\(...\)](#)  
x. [\\_\\_gt\\_\\_\(y\)](#) <==> x>y

[\\_\\_hash\\_\\_\(...\)](#)  
x. [\\_\\_hash\\_\\_\(\)](#) <==> hash(x)

[\\_\\_le\\_\\_\(...\)](#)  
x. [\\_\\_le\\_\\_\(y\)](#) <==> x<=y

[\\_\\_lt\\_\\_\(...\)](#)  
x. [\\_\\_lt\\_\\_\(y\)](#) <==> x<y

[\\_\\_ne\\_\\_\(...\)](#)  
x. [\\_\\_ne\\_\\_\(y\)](#) <==> x!=y

[\\_\\_repr\\_\\_\(...\)](#)  
x. [\\_\\_repr\\_\\_\(\)](#) <==> repr(x)

[\\_\\_str\\_\\_\(...\)](#)  
x. [\\_\\_str\\_\\_\(\)](#) <==> [str\(x\)](#)

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_\(S, ...\)](#) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [generic](#):

[\\_\\_abs\\_\\_\(...\)](#)  
x. [\\_\\_abs\\_\\_\(\)](#) <==> abs(x)

[\\_\\_add\\_\\_\(...\)](#)  
x. [\\_\\_add\\_\\_\(y\)](#) <==> x+y

[\\_\\_and\\_\\_\(...\)](#)  
x. [\\_\\_and\\_\\_\(y\)](#) <==> x&y

[\\_\\_array\\_\\_\(...\)](#)  
sc. [\\_\\_array\\_\\_\(\[type\]\)](#) return 0-dim array

[\\_\\_array\\_wrap\\_\\_\(...\)](#)  
sc. [\\_\\_array\\_wrap\\_\\_\(obj\)](#) return scalar from array

[\\_\\_copy\\_\\_\(...\)](#)

[\\_\\_deepcopy\\_\\_\(...\)](#)

```
div(...)
x._div_(y) <==> x/y

divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x
```

```
rlshift(...)
 x.rlshift(y) <==> y<<x

rmod(...)
 x.rmod(y) <==> y%x

rmul(...)
 x.rmul(y) <==> y*x

ror(...)
 x.ror(y) <==> y|x

rpow(...)
 y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
 x.rrshift(y) <==> y>>x

rshift(...)
 x.rshift(y) <==> x>>y

rsub(...)
 x.rsub(y) <==> y-x

rtruediv(...)
 x.rtruediv(y) <==> y/x

rxor(...)
 x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

sub(...)
 x.sub(y) <==> x-y

truediv(...)
 x.truediv(y) <==> x/y

xor(...)
 x.xor(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
```

```
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

**singlecomplex** = class complex64([complexfloating](#))

Composed of two 32 bit floats

Method resolution order:

[complex64](#)  
[complexfloating](#)  
[inexact](#)  
[number](#)  
[generic](#)  
[builtin\\_object](#)

Methods defined here:

[\\_\\_complex\\_\\_\(...\)](#)

[\\_\\_eq\\_\\_\(...\)](#)  
x. [\\_\\_eq\\_\\_\(y\)](#) <==> x==y

[\\_\\_ge\\_\\_\(...\)](#)  
x. [\\_\\_ge\\_\\_\(y\)](#) <==> x>=y

[\\_\\_gt\\_\\_\(...\)](#)  
x. [\\_\\_gt\\_\\_\(y\)](#) <==> x>y

[\\_\\_hash\\_\\_\(...\)](#)  
x. [\\_\\_hash\\_\\_\(\)](#) <==> hash(x)

[\\_\\_le\\_\\_\(...\)](#)  
x. [\\_\\_le\\_\\_\(y\)](#) <==> x<=y

[\\_\\_lt\\_\\_\(...\)](#)  
x. [\\_\\_lt\\_\\_\(y\)](#) <==> x<y

[\\_\\_ne\\_\\_\(...\)](#)  
x. [\\_\\_ne\\_\\_\(y\)](#) <==> x!=y

[\\_\\_repr\\_\\_\(...\)](#)  
x. [\\_\\_repr\\_\\_\(\)](#) <==> repr(x)

[\\_\\_str\\_\\_\(...\)](#)  
x. [\\_\\_str\\_\\_\(\)](#) <==> [str\(x\)](#)

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method \_\_new\_\_ of type object>  
T. [\\_\\_new\\_\\_\(S, ...\)](#) -> a new [object](#) with type S, a subtype of T

Methods inherited from [generic](#):

[\\_\\_abs\\_\\_\(...\)](#)  
x. [\\_\\_abs\\_\\_\(\)](#) <==> abs(x)

[\\_\\_add\\_\\_\(...\)](#)  
x. [\\_\\_add\\_\\_\(y\)](#) <==> x+y

```
__and__(...)
x.__and__(y) <==> x&y

__array__(...)
sc.__array__(|type) return 0-dim array

__array_wrap__(...)
sc.__array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x.__div__(y) <==> x/y

__divmod__(...)
x.__divmod__(y) <==> divmod(x, y)

__float__(...)
x.__float__() <==> float(x)

__floordiv__(...)
x.__floordiv__(y) <==> x//y

__format__(...)
NumPy array scalar formatter

__getitem__(...)
x.__getitem__(y) <==> x[y]

__hex__(...)
x.__hex__() <==> hex(x)

__int__(...)
x.__int__() <==> int(x)

__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
```

```
x. rand_(y) <==> y&x
rdiv_(...)
x. rdiv_(y) <==> y/x
rdivmod_(...)
x. rdivmod_(y) <==> divmod(y, x)
reduce_(...)
rfloordiv_(...)
x. rfloordiv_(y) <==> y//x
rlshift_(...)
x. rlshift_(y) <==> y<<x
rmod_(...)
x. rmod_(y) <==> y%x
rmul_(...)
x. rmul_(y) <==> y*x
ror_(...)
x. ror_(y) <==> y|x
rpow_(...)
y. rpow_(x[, z]) <==> pow(x, y[, z])
rrshift_(...)
x. rrshift_(y) <==> y>>x
rshift_(...)
x. rshift_(y) <==> x>>y
rsub_(...)
x. rsub_(y) <==> y-x
rtruediv_(...)
x. rtruediv_(y) <==> y/x
rxor_(...)
x. rxor_(y) <==> y^x
setstate_(...)
sizeof_(...)
sub_(...)
x. sub_(y) <==> x-y
truediv_(...)
x. truediv_(y) <==> x/y
xor_(...)
x. xor_(y) <==> x^y
all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
conj(...)
conjugate(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

copy(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

cumprod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

cumsum(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

diagonal(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

dump(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

dumps(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

fill(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

```
newbyteorder(...)
 newbyteorder\(new_order='S'\)

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 Parameters

 new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**taked(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### trace(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### transpose(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### var(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### view(...)

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

## Data descriptors inherited from [generic](#):

### T

    transpose

### \_\_array\_interface\_\_

    Array protocol: Python side

### \_\_array\_priority\_\_

    Array priority.

### \_\_array\_struct\_\_

    Array protocol: struct

### base

    base object

### data

    pointer to start of data

### dtype

    get array data-descriptor

### flags

    integer value of flags

### flat

```
a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

str_ = class string_(builtin_.str, character)
```

Method resolution order:

```
string
 builtin__str
 builtin__basestring
 character
 flexible
 generic
 builtin__object
```

---

Methods defined here:

```
__eq__\(...\)
 x.__eq__(y) <==> x==y

__ge__\(...\)
 x.__ge__(y) <==> x>=y

__gt__\(...\)
 x.__gt__(y) <==> x>y

__hash__\(...\)
 x.__hash__() <==> hash(x)

__le__\(...\)
 x.__le__(y) <==> x<=y

__lt__\(...\)
 x.__lt__(y) <==> x<y

__ne__\(...\)
 x.__ne__(y) <==> x!=y

__repr__\(...\)
 x.__repr__() <==> repr(x)

__str__\(...\)
 x.__str__() <==> str(x)
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

---

Methods inherited from [builtin\\_\\_str](#):

```
__add__\(...\)
 x.__add__(y) <==> x+y

__contains__\(...\)
 x.__contains__(y) <==> y in x

__format__\(...\)
 S.__format__(format_spec) -> string
 Return a formatted version of S as described by format_spec.

__getattribute__\(...\)
 x.__getattribute__('name') <==> x.name

__getitem__\(...\)
 x.__getitem__(y) <==> x[y]

__getnewargs__\(...\)

__getslice__\(...\)
 x.__getslice__(i, j) <==> x[i:j]
 Use of negative indices is not supported.

__len__\(...\)
 x.__len__() <==> len(x)
```

```
__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(n) <==> x*n

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(n) <==> n*x

sizeof(...)
 S.sizeof() -> size of S in memory, in bytes

capitalize(...)
 S.capitalize() -> string

 Return a copy of the string S with only its first character
 capitalized.

center(...)
 S.center(width[, fillchar]) -> string

 Return S centered in a string of length width. Padding is
 done using the specified fill character (default is a space)

count(...)
 S.count(sub[, start[, end]]) -> int

 Return the number of non-overlapping occurrences of substring sub in
 string S[start:end]. Optional arguments start and end are interpreted
 as in slice notation.

decode(...)
 S.decode([encoding[,errors]]) -> object

 Decodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'
 as well as any other name registered with codecs.register_error that is
 able to handle UnicodeDecodeErrors.

encode(...)
 S.encode([encoding[,errors]]) -> object

 Encodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
 'xmlcharrefreplace' as well as any other name registered with
 codecs.register_error that is able to handle UnicodeEncodeErrors.

endswith(...)
 S.endswith(suffix[, start[, end]]) -> bool

 Return True if S ends with the specified suffix, False otherwise.
 With optional start, test S beginning at that position.
 With optional end, stop comparing S at that position.
 suffix can also be a tuple of strings to try.

expandtabs(...)
 S.expandtabs([tabsize]) -> string

 Return a copy of S where all tab characters are expanded using spaces.
 If tabsize is not given, a tab size of 8 characters is assumed.

find(...)
 S.find(sub [,start [,end]]) -> int

 Return the lowest index in S where substring sub is found,
 such that sub is contained within S[start:end]. Optional
 arguments start and end are interpreted as in slice notation.

 Return -1 on failure.

format(...)
 S.format(*args, **kwargs) -> string

 Return a formatted version of S, using substitutions from args and kwargs.
```

The substitutions are identified by braces ('{' and '}').

**index(...)**  
S.index(sub [,start [,end]]) -> int  
Like S.find() but raise ValueError when the substring is not found.

**isalnum(...)**  
S.isalnum() -> bool  
Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

**isalpha(...)**  
S.isalpha() -> bool  
Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

**isdigit(...)**  
S.isdigit() -> bool  
Return True if all characters in S are digits and there is at least one character in S, False otherwise.

**islower(...)**  
S.islower() -> bool  
Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

**isspace(...)**  
S.isspace() -> bool  
Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

**istitle(...)**  
S.istitle() -> bool  
Return True if S is a titlecased string and there is at least one character in S, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

**isupper(...)**  
S.isupper() -> bool  
Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

**join(...)**  
S.join(iterable) -> string  
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

**ljust(...)**  
S.ljust(width[, fillchar]) -> string  
Return S left-justified in a string of length width. Padding is done using the specified fill character (default is a space).

**lower(...)**  
S.lower() -> string  
Return a copy of the string S converted to lowercase.

**lstrip(...)**  
S.lstrip([chars]) -> string or unicode  
Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode before stripping

**partition(...)**  
S.partition(sep) -> (head, sep, tail)  
Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

**replace(...)**  
S.replace(old, new[, count]) -> string  
  
Return a copy of string S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

**rfind(...)**  
S.rfind(sub [,start [,end]]) -> int  
  
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.  
  
Return -1 on failure.

**rindex(...)**  
S.rindex(sub [,start [,end]]) -> int  
  
Like S.rfind() but raise ValueError when the substring is not found.

**rjust(...)**  
S.rjust(width[, fillchar]) -> string  
  
Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space)

**rpartition(...)**  
S.rpartition(sep) -> (head, sep, tail)  
  
Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

**rsplit(...)**  
S.rsplit([sep [,maxsplit]]) -> list of strings  
  
Return a list of the words in the string S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator.

**rstrip(...)**  
S.rstrip([chars]) -> string or unicode  
  
Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode before stripping

**split(...)**  
S.split([sep [,maxsplit]]) -> list of strings  
  
Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

**splitlines(...)**  
S.splitlines(keepends=False) -> list of strings  
  
Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

**startswith(...)**  
S.startswith(prefix[, start[, end]]) -> bool  
  
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

**strip(...)**  
S.strip([chars]) -> string or unicode  
  
Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode before stripping

```
swapcase(...)
S.swapcase() -> string

 Return a copy of the string S with uppercase characters
 converted to lowercase and vice versa.

title(...)
S.title() -> string

 Return a titlecased version of S, i.e. words start with uppercase
 characters, all remaining cased characters have lowercase.

translate(...)
S.translate(table [,deletechars]) -> string

 Return a copy of the string S, where all characters occurring
 in the optional argument deletechars are removed, and the
 remaining characters have been mapped through the given
 translation table, which must be a string of length 256 or None.
 If the table argument is None, no translation is applied and
 the operation simply removes the characters in deletechars.

upper(...)
S.upper() -> string

 Return a copy of the string S converted to uppercase.

zfill(...)
S.zfill(width) -> string

 Pad a numeric string S with zeros on the left, to fill a field
 of the specified width. The string S is never truncated.
```

---

Methods inherited from [generic](#):

```
_abs(...)
x.abs() <==> abs(x)

_and(...)
x.and(y) <==> x&y

_array(...)
sc.array(|type) return 0-dim array

_array_wrap(...)
sc.array_wrap(obj) return scalar from array

_copy(...)

_deepcopy(...)

_div(...)
x.div(y) <==> x/y

_divmod(...)
x.divmod(y) <==> divmod(x, y)

_float(...)
x.float() <==> float(x)

_floordiv(...)
x.floordiv(y) <==> x//y

_hex(...)
x.hex() <==> hex(x)

_int(...)
x.int() <==> int(x)

_invert(...)
x.invert() <==> ~x

_long(...)
x.long() <==> long(x)

_lshift(...)
```

```
x. lshift_(y) <==> x<<y
neg_(...)
x. neg_(y) <==> -x
nonzero_(...)
x. nonzero_(y) <==> x != 0
oct_(...)
x. oct_(y) <==> oct(x)
or_(...)
x. or_(y) <==> x|y
pos_(...)
x. pos_(y) <==> +x
pow_(...)
x. pow_(y[, z]) <==> pow(x, y[, z])
radd_(...)
x. radd_(y) <==> y+x
rand_(...)
x. rand_(y) <==> y&x
rdiv_(...)
x. rdiv_(y) <==> y/x
rdivmod_(...)
x. rdivmod_(y) <==> divmod(y, x)
reduce_(...)
rfloordiv_(...)
x. rfloordiv_(y) <==> y//x
rlshift_(...)
x. rlshift_(y) <==> y<<x
ror_(...)
x. ror_(y) <==> y|x
rpow_(...)
y. rpow_(x[, z]) <==> pow(x, y[, z])
rrshift_(...)
x. rrshift_(y) <==> y>>x
rshift_(...)
x. rshift_(y) <==> x>>y
rsub_(...)
x. rsub_(y) <==> y-x
rtruediv_(...)
x. rtruediv_(y) <==> y/x
rxor_(...)
x. rxor_(y) <==> y^x
setstate_(...)
sub_(...)
x. sub_(y) <==> x-y
truediv_(...)
x. truediv_(y) <==> x/y
xor_(...)
x. xor_(y) <==> x^y
all(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_array\_interface\_**  
Array protocol: Python side

**\_array\_priority\_**  
Array priority.

```
__array_struct__
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

---

```
string0 = class string_(_builtin_.str, character)
```

Method resolution order:

```
string
 builtin.str
 builtin.basestring
character
flexible
generic
 builtin.object
```

---

Methods defined here:

```
__eq__(...)
 x. __eq__(y) <==> x==y

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__hash__(...)
 x. __hash__() <==> hash(x)

__le__(...)
 x. __le__(y) <==> x<=y

__lt__(...)
```

---

```

x.lt(y) <==> x<y

ne(...)
x.ne(y) <==> x!=y

repr(...)
x.repr() <==> repr(x)

str(...)
x.str() <==> str(x)

```

---

Data and other attributes defined here:

new = <built-in method new of type object>  
T.new(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from builtin.str:

```

add(...)
x.add(y) <==> x+y

contains(...)
x.contains(y) <==> y in x

format(...)
S.format(format_spec) -> string

 Return a formatted version of S as described by format_spec.

getattribute(...)
x.getattribute('name') <==> x.name

getitem(...)
x.getitem(y) <==> x[y]

getnewargs(...)

getslice(...)
x.getslice(i, j) <==> x[i:j]

 Use of negative indices is not supported.

len(...)
x.len() <==> len(x)

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(n) <==> x*n

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(n) <==> n*x

sizeof(...)
S.sizeof() -> size of S in memory, in bytes

capitalize(...)
S.capitalize() -> string

 Return a copy of the string S with only its first character
capitalized.

center(...)
S.center(width[, fillchar]) -> string

 Return S centered in a string of length width. Padding is
done using the specified fill character (default is a space)

count(...)
S.count(sub[, start[, end]]) -> int

```

Return the `number` of non-overlapping occurrences of substring `sub` in string `S[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

**decode(...)**  
`S.decode([encoding[,errors]]) -> object`

Decodes `S` using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a `UnicodeDecodeError`. Other possible values are 'ignore' and 'replace' as well as any other name registered with `codecs.register_error` that is able to handle `UnicodeDecodeErrors`.

**encode(...)**  
`S.encode([encoding[,errors]]) -> object`

Encodes `S` using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with `codecs.register_error` that is able to handle `UnicodeEncodeErrors`.

**endswith(...)**  
`S.endswith(suffix[, start[, end]]) -> bool`

Return True if `S` ends with the specified suffix, False otherwise.  
With optional `start`, test `S` beginning at that position.  
With optional `end`, stop comparing `S` at that position.  
suffix can also be a tuple of strings to try.

**expandtabs(...)**  
`S.expandtabs([tabsize]) -> string`

Return a copy of `S` where all tab characters are expanded using spaces.  
If `tabsize` is not given, a tab size of 8 characters is assumed.

**find(...)**  
`S.find(sub [,start [,end]]) -> int`

Return the lowest index in `S` where substring `sub` is found, such that `sub` is contained within `S[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Return -1 on failure.

**format(...)**  
`S.format(*args, **kwargs) -> string`

Return a formatted version of `S`, using substitutions from `args` and `kwargs`.  
The substitutions are identified by braces ('{' and '}').

**index(...)**  
`S.index(sub [,start [,end]]) -> int`

Like `S.find()` but raise `ValueError` when the substring is not found.

**isalnum(...)**  
`S.isalnum() -> bool`

Return True if all characters in `S` are alphanumeric and there is at least one `character` in `S`, False otherwise.

**isalpha(...)**  
`S.isalpha() -> bool`

Return True if all characters in `S` are alphabetic and there is at least one `character` in `S`, False otherwise.

**isdigit(...)**  
`S.isdigit() -> bool`

Return True if all characters in `S` are digits and there is at least one `character` in `S`, False otherwise.

**islower(...)**  
`S.islower() -> bool`

Return True if all cased characters in `S` are lowercase and there is at least one cased `character` in `S`, False otherwise.

**isspace(...)**  
S.isspace() -> bool  
Return True if all characters in S are whitespace  
and there is at least one character in S, False otherwise.

**istitle(...)**  
S.istitle() -> bool  
Return True if S is a titlecased string and there is at least one  
character in S, i.e. uppercase characters may only follow uncased  
characters and lowercase characters only cased ones. Return False  
otherwise.

**isupper(...)**  
S.isupper() -> bool  
Return True if all cased characters in S are uppercase and there is  
at least one cased character in S, False otherwise.

**join(...)**  
S.join(iterable) -> string  
Return a string which is the concatenation of the strings in the  
iterable. The separator between elements is S.

**ljust(...)**  
S.ljust(width[, fillchar]) -> string  
Return S left-justified in a string of length width. Padding is  
done using the specified fill character (default is a space).

**lower(...)**  
S.lower() -> string  
Return a copy of the string S converted to lowercase.

**lstrip(...)**  
S.lstrip([chars]) -> string or unicode  
Return a copy of the string S with leading whitespace removed.  
If chars is given and not None, remove characters in chars instead.  
If chars is unicode, S will be converted to unicode before stripping

**partition(...)**  
S.partition(sep) -> (head, sep, tail)  
Search for the separator sep in S, and return the part before it,  
the separator itself, and the part after it. If the separator is not  
found, return S and two empty strings.

**replace(...)**  
S.replace(old, new[, count]) -> string  
Return a copy of string S with all occurrences of substring  
old replaced by new. If the optional argument count is  
given, only the first count occurrences are replaced.

**rfind(...)**  
S.rfind(sub [,start [,end]]) -> int  
Return the highest index in S where substring sub is found,  
such that sub is contained within S[start:end]. Optional  
arguments start and end are interpreted as in slice notation.  
Return -1 on failure.

**rindex(...)**  
S.rindex(sub [,start [,end]]) -> int  
Like S.rfind() but raise ValueError when the substring is not found.

**rjust(...)**  
S.rjust(width[, fillchar]) -> string  
Return S right-justified in a string of length width. Padding is  
done using the specified fill character (default is a space)

**rpartition(...)**  
S.rpartition(sep) -> (head, sep, tail)  
Search for the separator sep in S, starting at the end of S, and return

the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

**rsplit(...)**  
S.[rsplit](#)([sep [,maxsplit]]) -> list of strings  
  
Return a list of the words in the string S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator.

**rstrip(...)**  
S.[rstrip](#)([chars]) -> string or [unicode](#)  
  
Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is [unicode](#), S will be converted to [unicode](#) before stripping

**split(...)**  
S.[split](#)([sep [,maxsplit]]) -> list of strings  
  
Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

**splitlines(...)**  
S.[splitlines](#)(keepends=False) -> list of strings  
  
Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

**startswith(...)**  
S.[startswith](#)(prefix[, start[, end]]) -> bool  
  
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

**strip(...)**  
S.[strip](#)([chars]) -> string or [unicode](#)  
  
Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is [unicode](#), S will be converted to [unicode](#) before stripping

**swapcase(...)**  
S.[swapcase](#)() -> string  
  
Return a copy of the string S with uppercase characters converted to lowercase and vice versa.

**title(...)**  
S.[title](#)() -> string  
  
Return a titlecased version of S, i.e. words start with uppercase characters, all remaining cased characters have lowercase.

**translate(...)**  
S.[translate](#)(table [,deletechars]) -> string  
  
Return a copy of the string S, where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256 or None. If the table argument is None, no translation is applied and the operation simply removes the characters in deletechars.

**upper(...)**  
S.[upper](#)() -> string  
  
Return a copy of the string S converted to uppercase.

**zfill(...)**  
S.[zfill](#)(width) -> string  
  
Pad a numeric string S with zeros on the left, to fill a field

---

of the specified width. The string S is never truncated.

---

Methods inherited from [generic](#):

[abs](#)(...)  
x.[abs](#)() <==> abs(x)

[and](#)(...)  
x.[and](#)(y) <==> x&y

[array](#)(...)  
sc.[array](#)(|type) return 0-dim array

[array\\_wrap](#)(...)  
sc.[array\\_wrap](#)(obj) return scalar from array

[copy](#)(...)

[deepcopy](#)(...)

[div](#)(...)  
x.[div](#)(y) <==> x/y

[divmod](#)(...)  
x.[divmod](#)(y) <==> divmod(x, y)

[float](#)(...)  
x.[float](#)() <==> float(x)

[floordiv](#)(...)  
x.[floordiv](#)(y) <==> x//y

[hex](#)(...)  
x.[hex](#)() <==> hex(x)

[int](#)(...)  
x.[int](#)() <==> int(x)

[invert](#)(...)  
x.[invert](#)() <==> ~x

[long](#)(...)  
x.[long](#)() <==> long(x)

[lshift](#)(...)  
x.[lshift](#)(y) <==> x<<y

[neg](#)(...)  
x.[neg](#)() <==> -x

[nonzero](#)(...)  
x.[nonzero](#)() <==> x != 0

[oct](#)(...)  
x.[oct](#)() <==> oct(x)

[or](#)(...)  
x.[or](#)(y) <==> x|y

[pos](#)(...)  
x.[pos](#)() <==> +x

[pow](#)(...)  
x.[pow](#)(y[, z]) <==> pow(x, y[, z])

[radd](#)(...)  
x.[radd](#)(y) <==> y+x

[rand](#)(...)  
x.[rand](#)(y) <==> y&x

[rdiv](#)(...)  
x.[rdiv](#)(y) <==> y/x

```
__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
```

```
* {'=': 'N'} - native order
* {'|': 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

#### **T**

transpose

#### **\_\_array\_interface\_\_**

Array protocol: Python side

#### **\_\_array\_priority\_\_**

Array priority.

#### **\_\_array\_struct\_\_**

Array protocol: struct

#### **base**

base object

#### **data**

pointer to start of data

#### **dtype**

get array data-descriptor

#### **flags**

integer value of flags

#### **flat**

a 1-d view of scalar

#### **imag**

imaginary part of scalar

#### **itemsize**

length of one element in bytes

#### **nbytes**

length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

class **string\_**([\\_builtin\\_.str](#), [character](#))

Method resolution order:

[string](#)  
[\\_builtin\\_.str](#)  
[\\_builtin\\_.basestring](#)  
[character](#)  
[flexible](#)  
[generic](#)  
[\\_builtin\\_.object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_\(...\)](#)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_ge\\_\\_\(...\)](#)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_\(...\)](#)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_\(...\)](#)  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_le\\_\\_\(...\)](#)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_lt\\_\\_\(...\)](#)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_\(...\)](#)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

[\\_\\_repr\\_\\_\(...\)](#)  
x. [\\_\\_repr\\_\\_](#)() <==> repr(x)

[\\_\\_str\\_\\_\(...\)](#)  
x. [\\_\\_str\\_\\_](#)() <==> [str](#)(x)

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Methods inherited from [\\_builtin\\_.str](#):

[\\_\\_add\\_\\_\(...\)](#)  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y

[\\_\\_contains\\_\\_\(...\)](#)  
x. [\\_\\_contains\\_\\_](#)(y) <==> y in x

[\\_\\_format\\_\\_\(...\)](#)  
S. [\\_\\_format\\_\\_](#)(format\_spec) -> string

```
 Return a formatted version of S as described by format_spec.

__getattribute__(...)
 x.getattribute('name') <==> x.name

__getitem__(...)
 x.getitem(y) <==> x[y]

__getnewargs__(...)

__getslice__(...)
 x.getslice(i, j) <==> x[i:j]

 Use of negative indices is not supported.

__len__(...)
 x.len() <==> len(x)

__mod__(...)
 x.mod(y) <==> x%y

__mul__(...)
 x.mul(n) <==> x*n

__rmod__(...)
 x.rmod(y) <==> y%x

__rmul__(...)
 x.rmul(n) <==> n*x

__sizeof__(...)
 S.sizeof() -> size of S in memory, in bytes

capitalize(...)
 S.capitalize() -> string

 Return a copy of the string S with only its first character
 capitalized.

center(...)
 S.center(width[, fillchar]) -> string

 Return S centered in a string of length width. Padding is
 done using the specified fill character (default is a space)

count(...)
 S.count(sub[, start[, end]]) -> int

 Return the number of non-overlapping occurrences of substring sub in
 string S[start:end]. Optional arguments start and end are interpreted
 as in slice notation.

decode(...)
 S.decode([encoding[,errors]]) -> object

 Decodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'
 as well as any other name registered with codecs.register_error that is
 able to handle UnicodeDecodeErrors.

encode(...)
 S.encode([encoding[,errors]]) -> object

 Encodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
 'xmlcharrefreplace' as well as any other name registered with
 codecs.register_error that is able to handle UnicodeEncodeErrors.

endswith(...)
 S.endswith(suffix[, start[, end]]) -> bool

 Return True if S ends with the specified suffix, False otherwise.
 With optional start, test S beginning at that position.
 With optional end, stop comparing S at that position.
```

suffix can also be a tuple of strings to try.

**expandtabs(...)**  
S.[expandtabs](#)([tabsize]) -> string  
Return a copy of S where all tab characters are expanded using spaces.  
If tabsize is not given, a tab size of 8 characters is assumed.

**find(...)**  
S.[find](#)(sub [,start [,end]]) -> [int](#)  
Return the lowest index in S where substring sub is found,  
such that sub is contained within S[start:end]. Optional  
arguments start and end are interpreted as in slice notation.  
Return -1 on failure.

**format(...)**  
S.[format](#)(\*args, \*\*kwargs) -> string  
Return a formatted version of S, using substitutions from args and kwargs.  
The substitutions are identified by braces ('{' and '}').

**index(...)**  
S.[index](#)(sub [,start [,end]]) -> [int](#)  
Like S.[find\(\)](#) but raise ValueError when the substring is not found.

**isalnum(...)**  
S.[isalnum](#)() -> bool  
Return True if all characters in S are alphanumeric  
and there is at least one [character](#) in S, False otherwise.

**isalpha(...)**  
S.[isalpha](#)() -> bool  
Return True if all characters in S are alphabetic  
and there is at least one [character](#) in S, False otherwise.

**isdigit(...)**  
S.[isdigit](#)() -> bool  
Return True if all characters in S are digits  
and there is at least one [character](#) in S, False otherwise.

**islower(...)**  
S.[islower](#)() -> bool  
Return True if all cased characters in S are lowercase and there is  
at least one cased [character](#) in S, False otherwise.

**isspace(...)**  
S.[isspace](#)() -> bool  
Return True if all characters in S are whitespace  
and there is at least one [character](#) in S, False otherwise.

**istitle(...)**  
S.[istitle](#)() -> bool  
Return True if S is a titlecased string and there is at least one  
[character](#) in S, i.e. uppercase characters may only follow uncased  
characters and lowercase characters only cased ones. Return False  
otherwise.

**isupper(...)**  
S.[isupper](#)() -> bool  
Return True if all cased characters in S are uppercase and there is  
at least one cased [character](#) in S, False otherwise.

**join(...)**  
S.[join](#)(iterable) -> string  
Return a string which is the concatenation of the strings in the  
iterable. The separator between elements is S.

**ljust(...)**  
S.[ljust](#)(width[, fillchar]) -> string

Return S left-justified in a string of length width. Padding is done using the specified fill [character](#) (default is a space).

**lower(...)**  
S.[lower](#)() -> string  
  
Return a copy of the string S converted to lowercase.

**lstrip(...)**  
S.[lstrip](#)([chars]) -> string or [unicode](#)  
  
Return a copy of the string S with leading whitespace removed.  
If chars is given and not None, remove characters in chars instead.  
If chars is [unicode](#), S will be converted to [unicode](#) before stripping

**partition(...)**  
S.[partition](#)(sep) -> (head, sep, tail)  
  
Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

**replace(...)**  
S.[replace](#)(old, new[, count]) -> string  
  
Return a copy of string S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

**rfind(...)**  
S.[rfind](#)(sub [,start [,end]]) -> [int](#)  
  
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.  
  
Return -1 on failure.

**rindex(...)**  
S.[rindex](#)(sub [,start [,end]]) -> [int](#)  
  
Like S.[rfind\(\)](#) but raise ValueError when the substring is not found.

**rjust(...)**  
S.[rjust](#)(width[, fillchar]) -> string  
  
Return S right-justified in a string of length width. Padding is done using the specified fill [character](#) (default is a space)

**rpartition(...)**  
S.[rpartition](#)(sep) -> (head, sep, tail)  
  
Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

**rsplit(...)**  
S.[rsplit](#)([sep [,maxsplit]]) -> list of strings  
  
Return a list of the words in the string S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator.

**rstrip(...)**  
S.[rstrip](#)([chars]) -> string or [unicode](#)  
  
Return a copy of the string S with trailing whitespace removed.  
If chars is given and not None, remove characters in chars instead.  
If chars is [unicode](#), S will be converted to [unicode](#) before stripping

**split(...)**  
S.[split](#)([sep [,maxsplit]]) -> list of strings  
  
Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

**splitlines(...)**

```

S.splitlines(keepends=False) -> list of strings
 Return a list of the lines in S, breaking at line boundaries.
 Line breaks are not included in the resulting list unless keepends
 is given and true.

startswith(...)
 S.startswith(prefix[, start[, end]]) -> bool
 Return True if S starts with the specified prefix, False otherwise.
 With optional start, test S beginning at that position.
 With optional end, stop comparing S at that position.
 prefix can also be a tuple of strings to try.

strip(...)
 S.strip([chars]) -> string or unicode
 Return a copy of the string S with leading and trailing
 whitespace removed.
 If chars is given and not None, remove characters in chars instead.
 If chars is unicode, S will be converted to unicode before stripping

swapcase(...)
 S.swapcase() -> string
 Return a copy of the string S with uppercase characters
 converted to lowercase and vice versa.

title(...)
 S.title() -> string
 Return a titlecased version of S, i.e. words start with uppercase
 characters, all remaining cased characters have lowercase.

translate(...)
 S.translate(table [,deletechars]) -> string
 Return a copy of the string S, where all characters occurring
 in the optional argument deletechars are removed, and the
 remaining characters have been mapped through the given
 translation table, which must be a string of length 256 or None.
 If the table argument is None, no translation is applied and
 the operation simply removes the characters in deletechars.

upper(...)
 S.upper() -> string
 Return a copy of the string S converted to uppercase.

zfill(...)
 S.zfill(width) -> string
 Pad a numeric string S with zeros on the left, to fill a field
 of the specified width. The string S is never truncated.

```

---

Methods inherited from [generic](#):

```

__abs__(...)
 x.abs() <==> abs(x)

__and__(...)
 x.and(y) <==> x&y

__array__(...)
 sc.array(|type) return 0-dim array

__array_wrap__(...)
 sc.array_wrap(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
 x.div(y) <==> x/y

__divmod__(...)
 x.divmod(y) <==> divmod(x, y)

```

```
float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

ror(...)
x._ror_(y) <==> y|x

rpow(...)
y._rpow_(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x._rrshift_(y) <==> y>>x

rshift(...)
x._rshift_(y) <==> x>>y

rsub(...)
x._rsub_(y) <==> y-x
```

**\_\_rtruediv\_\_(...)**  
x. rtruediv(y) <==> y/x

**\_\_rxor\_\_(...)**  
x. rxor(y) <==> y^x

**\_\_setstate\_\_(...)**

**\_\_sub\_\_(...)**  
x. sub(y) <==> x-y

**\_\_truediv\_\_(...)**  
x. truediv(y) <==> x/y

**\_\_xor\_\_(...)**  
x. xor(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

 The corresponding attribute of the derived class of interest.

take(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tolist(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

toString(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

    transpose

**\_\_array\_interface\_\_**

    Array protocol: Python side

**\_\_array\_priority\_\_**

    Array priority.

**\_\_array\_struct\_\_**

    Array protocol: struct

**base**

    base object

**data**

    pointer to start of data

**dtype**

    get array data-descriptor

**flags**

    integer value of flags

**flat**

    a 1-d view of scalar

**imag**

    imaginary part of scalar

**itemsize**

    length of one element in bytes

**nbytes**

    length of item in bytes

**ndim**

    number of array dimensions

**real**

    real part of scalar

**shape**

    tuple of array dimensions

**size**

    number of elements in the gentype

**strides**

    tuple of bytes steps in each dimension

---

class **timedelta64**([signedinteger](#))

Method resolution order:

**timedelta64**  
    [signedinteger](#)  
    [integer](#)  
    [number](#)  
    [generic](#)  
    [builtin](#) .[object](#)

---

Methods defined here:

---

**`__eq__(...)`**  
x. `__eq__(y) <==> x==y`

**`__ge__(...)`**  
x. `__ge__(y) <==> x>=y`

**`__gt__(...)`**  
x. `__gt__(y) <==> x>y`

**`__hash__(...)`**  
x. `__hash__() <==> hash(x)`

**`__le__(...)`**  
x. `__le__(y) <==> x<=y`

**`__lt__(...)`**  
x. `__lt__(y) <==> x<y`

**`__ne__(...)`**  
x. `__ne__(y) <==> x!=y`

**`__repr__(...)`**  
x. `__repr__() <==> repr(x)`

**`__str__(...)`**  
x. `__str__() <==> str(x)`

---

Data and other attributes defined here:

---

**`__new__`** = <built-in method `__new__` of type object>  
T. `__new__(S, ...)` -> a new `object` with type S, a subtype of T

---

Data descriptors inherited from `integer`:

**`denominator`**  
denominator of value (1)

**`numerator`**  
numerator of value (the value itself)

---

Methods inherited from `generic`:

---

**`__abs__(...)`**  
x. `__abs__() <==> abs(x)`

**`__add__(...)`**  
x. `__add__(y) <==> x+y`

**`__and__(...)`**  
x. `__and__(y) <==> x&y`

**`__array__(...)`**  
sc. `__array__(|type|) return 0-dim array`

**`__array_wrap__(...)`**  
sc. `__array_wrap__(obj) return scalar from array`

**`__copy__(...)`**

**`__deepcopy__(...)`**

**`__div__(...)`**  
x. `__div__(y) <==> x/y`

**`__divmod__(...)`**  
x. `__divmod__(y) <==> divmod(x, y)`

**`__float__(...)`**  
x. `__float__() <==> float(x)`

```
floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

rmod(...)
x._rmod_(y) <==> y%x

rmul(...)
x._rmul_(y) <==> y*x
```

```
__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.
```

Returns  
-----  
**new\_dtype** : [dtype](#)  
    New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_array\_interface\_**  
Array protocol: Python side

**\_array\_priority\_**  
Array priority.

**\_array\_struct\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**

number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

---

**ubyte** = class uint8([unsignedinteger](#))

Method resolution order:

[uint8](#)  
[unsignedinteger](#)  
[integer](#)  
[number](#)  
[generic](#)  
[builtin](#) [object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_](#)(...)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y  
  
[\\_\\_ge\\_\\_](#)(...)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y  
  
[\\_\\_gt\\_\\_](#)(...)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y  
  
[\\_\\_hash\\_\\_](#)(...)  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)  
  
[\\_\\_index\\_\\_](#)(...)  
x[y:z] <==> x[y. [\\_\\_index\\_\\_](#)():z. [\\_\\_index\\_\\_](#)()]  
  
[\\_\\_le\\_\\_](#)(...)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y  
  
[\\_\\_lt\\_\\_](#)(...)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y  
  
[\\_\\_ne\\_\\_](#)(...)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

---

Methods inherited from [generic](#):

[\\_\\_abs\\_\\_](#)(...)  
x. [\\_\\_abs\\_\\_](#)() <==> abs(x)  
  
[\\_\\_add\\_\\_](#)(...)  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y  
  
[\\_\\_and\\_\\_](#)(...)  
x. [\\_\\_and\\_\\_](#)(y) <==> x&y  
  
[\\_\\_array\\_\\_](#)(...)  
sc. [\\_\\_array\\_\\_](#)(|type) return 0-dim array  
  
[\\_\\_array\\_wrap\\_\\_](#)(...)

```
sc.array_wrap(obj) return scalar from array
copy(...)
deepcopy(...)
div(...)
x.div(y) <==> x/y
divmod(...)
x.divmod(y) <==> divmod(x, y)
float(...)
x.float() <==> float(x)
floordiv(...)
x.floordiv(y) <==> x//y
format(...)
NumPy array scalar formatter
getitem(...)
x.getitem(y) <==> x[y]
hex(...)
x.hex() <==> hex(x)
int(...)
x.int() <==> int(x)
invert(...)
x.invert() <==> ~x
long(...)
x.long() <==> long(x)
lshift(...)
x.lshift(y) <==> x<<y
mod(...)
x.mod(y) <==> x%y
mul(...)
x.mul(y) <==> x*y
neg(...)
x.neg() <==> -x
nonzero(...)
x.nonzero() <==> x != 0
oct(...)
x.oct() <==> oct(x)
or(...)
x.or(y) <==> x|y
pos(...)
x.pos() <==> +x
pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])
radd(...)
x.radd(y) <==> y+x
rand(...)
x.rand(y) <==> y&x
rdiv(...)
x.rdiv(y) <==> y/x
rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)
```

```
__reduce__(...)

__repr__(...)
 x.__repr__() <==> repr(x)

__rfloordiv__(...)
 x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
 x.__rlshift__(y) <==> y<<x

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(y) <==> y*x

__ror__(...)
 x.__ror__(y) <==> y|x

__rpow__(...)
 y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x.__rrshift__(y) <==> y>>x

__rshift__(...)
 x.__rshift__(y) <==> x>>y

__rsub__(...)
 x.__rsub__(y) <==> y-x

__rtruediv__(...)
 x.__rtruediv__(y) <==> y/x

__rxor__(...)
 x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x.__str__() <==> str(x)

__sub__(...)
 x.__sub__(y) <==> x-y

__truediv__(...)
 x.__truediv__(y) <==> x/y

__xor__(...)
 x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

```
newbyteorder(...)
 newbyteorder(new_order='S')

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

 Parameters

 new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

 Returns

 new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**taked(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**

```
a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

class ufunc(builtin_.object)
 Functions that operate element by element on whole arrays.

 To see the documentation for a specific ufunc, use np.info(). For example, np.info(np.sin). Because ufuncs are written in C (for speed) and linked into Python with NumPy's ufunc facility, Python's help() function finds this page whenever help() is called on a ufunc.

 A detailed explanation of ufuncs can be found in the "ufuncs.rst" file in the NumPy reference guide.

 Unary ufuncs:
 =====
 op(X, out=None)
 Apply op to X elementwise

 Parameters

 X : array_like
 Input array.
 out : array_like
 An array to store the output. Must be the same shape as `X`.

 Returns

 r : array_like
 `r` will have the same shape as `X`; if out is provided, `r` will be equal to out.

 Binary ufuncs:
 =====
 op(X, Y, out=None)
 Apply `op` to `X` and `Y` elementwise. May "broadcast" to make the shapes of `X` and `Y` congruent.

 The broadcasting rules are:
 * Dimensions of length 1 may be prepended to either array.
 * Arrays may be repeated along dimensions of length 1.

 Parameters

 X : array_like
 First input array.
 Y : array_like
 Second input array.
 out : array_like
 An array to store the output. Must be the same shape as the output would have.

 Returns

```

```
r : array_like
 The return value; if out is provided, `r` will be equal to out.

Methods defined here:

__call__(...)
 x.__call__(...) <=> x(...)

__repr__(...)
 x.__repr__() <=> repr(x)

__str__(...)
 x.__str__() <=> str(x)

accumulate(...)
 accumulate(array, axis=0, dtype=None, out=None)

 Accumulate the result of applying the operator to all elements.

 For a one-dimensional array, accumulate produces results equivalent to::

 r = np.empty(len(A))
 t = op.identity # op = the ufunc being applied to A's elements
 for i in range(len(A)):
 t = op(t, A[i])
 r[i] = t
 return r

 For example, add.accumulate() is equivalent to np.cumsum().

 For a multi-dimensional array, accumulate is applied along only one
 axis (axis zero by default; see Examples below) so repeated use is
 necessary if one wants to accumulate over multiple axes.

Parameters

array : array_like
 The array to act on.
axis : int, optional
 The axis along which to apply the accumulation; default is zero.
dtype : data-type code, optional
 The data-type used to represent the intermediate results. Defaults
 to the data-type of the output array if such is provided, or the
 the data-type of the input array if no output array is provided.
out : ndarray, optional
 A location into which the result is stored. If not provided a
 freshly-allocated array is returned.

Returns

r : ndarray
 The accumulated values. If `out` was supplied, `r` is a reference to
 `out`.

Examples

1-D array examples:

>>> np.add.accumulate([2, 3, 5])
array([2, 5, 10])
>>> np.multiply.accumulate([2, 3, 5])
array([2, 6, 30])

2-D array examples:

>>> I = np.eye(2)
>>> I
array([[1., 0.],
 [0., 1.]})

 Accumulate along axis 0 (rows), down columns:

>>> np.add.accumulate(I, 0)
array([[1., 0.],
 [1., 1.]])
>>> np.add.accumulate(I) # no axis specified = axis zero
array([[1., 0.],
 [1., 1.]})

 Accumulate along axis 1 (columns), through rows:

>>> np.add.accumulate(I, 1)
array([[1., 1.],
 [0., 1.]])
```

**at(...)**

```
at(a, indices, b=None)
 Performs unbuffered in place operation on operand 'a' for elements
 specified by 'indices'. For addition ufunc, this method is equivalent to
 'a[indices] += b', except that results are accumulated for elements that
 are indexed more than once. For example, 'a[[0,0]] += 1' will only
 increment the first element once because of buffering, whereas
 'add.at(a, [0,0], 1)' will increment the first element twice.

 .. versionadded:: 1.8.0

Parameters

a : array_like
 The array to perform in place operation on.
indices : array_like or tuple
 Array like index object or slice object for indexing into first
 operand. If first operand has multiple dimensions, indices can be a
 tuple of array like index objects or slice objects.
b : array_like
 Second operand for ufuncs requiring two operands. Operand must be
 broadcastable over first operand after indexing or slicing.

Examples

Set items 0 and 1 to their negative values:

>>> a = np.array([1, 2, 3, 4])
>>> np.negative.at(a, [0, 1])
>>> print(a)
array([-1, -2, 3, 4])

::

Increment items 0 and 1, and increment item 2 twice:

>>> a = np.array([1, 2, 3, 4])
>>> np.add.at(a, [0, 1, 2, 2], 1)
>>> print(a)
array([2, 3, 5, 4])

::

Add items 0 and 1 in first array to second array,
and store results in first array:

>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 2])
>>> np.add.at(a, [0, 1], b)
>>> print(a)
array([2, 4, 3, 4])

outer(...)
outer(A, B)
 Apply the ufunc `op` to all pairs (a, b) with a in `A` and b in `B`.

 Let ``M = A.ndim``, ``N = B.ndim``. Then the result, `C`, of
 ``op.outer(A, B)`` is an array of dimension M + N such that:

 .. math::
 C[i_0, \dots, i_{M-1}, j_0, \dots, j_{N-1}] = op(A[i_0, \dots, i_{M-1}], B[j_0, \dots, j_{N-1}])

 For `A` and `B` one-dimensional, this is equivalent to:

 r = empty(len(A), len(B))
 for i in range(len(A)):
 for j in range(len(B)):
 r[i,j] = op(A[i], B[j]) # op = ufunc in question

Parameters

A : array_like
 First array
B : array_like
 Second array

Returns

r : ndarray
 Output array

See Also

numpy.outer

Examples

>>> np.multiply.outer([1, 2, 3], [4, 5, 6])
```

```

array([[4, 5, 6],
 [8, 10, 12],
 [12, 15, 18]])

A multi-dimensional example:

>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1, 2, 3, 4]])
>>> B.shape
(1, 4)
>>> C = np.multiply.outer(A, B)
>>> C.shape; C
(2, 3, 1, 4)
array([[[[1, 2, 3, 4]],
 [[2, 4, 6, 8]],
 [[3, 6, 9, 12]]],
 [[[4, 8, 12, 16]],
 [[5, 10, 15, 20]],
 [[6, 12, 18, 24]]]]))

reduce(...)
reduce(a, axis=0, dtype=None, out=None, keepdims=False)

Reduces `a`'s dimension by one, by applying ufunc along one axis.

Let :math:`a.shape = (N_0, \dots, N_i, \dots, N_{M-1})`. Then
:math:`\text{ufunc}.reduce(a, \text{axis}=i)[k_0, \dots, k_{i-1}, k_{i+1}, \dots, k_{M-1}]` =
the result of iterating `j` over :math:`\text{range}(N_i)` , cumulatively applying
ufunc to each :math:`a[k_0, \dots, k_{i-1}, j, k_{i+1}, \dots, k_{M-1}]`.
For a one-dimensional array, reduce produces results equivalent to:
::

r = op.identity # op = ufunc
for i in range(len(A)):
 r = op(r, A[i])
return r

For example, add.reduce() is equivalent to sum().

Parameters

a : array_like
 The array to act on.
axis : None or int or tuple of ints, optional
 Axis or axes along which a reduction is performed.
 The default ('axis' = 0) is perform a reduction over the first
 dimension of the input array. 'axis' may be negative, in
 which case it counts from the last to the first axis.

.. versionadded:: 1.7.0

If this is 'None', a reduction is performed over all the axes.
If this is a tuple of ints, a reduction is performed on multiple
axes, instead of a single axis or all the axes as before.

For operations which are either not commutative or not associative,
doing a reduction over multiple axes is not well-defined. The
ufuncs do not currently raise an exception in this case, but will
likely do so in the future.

dtype : data-type code, optional
 The type used to represent the intermediate results. Defaults
 to the data-type of the output array if this is provided, or
 the data-type of the input array if no output array is provided.

out : ndarray, optional
 A location into which the result is stored. If not provided, a
 freshly-allocated array is returned.

keepdims : bool, optional
 If this is set to True, the axes which are reduced are left
 in the result as dimensions with size one. With this option,
 the result will broadcast correctly against the original 'arr'.

.. versionadded:: 1.7.0

Returns

r : ndarray
 The reduced array. If 'out' was supplied, `r` is a reference to it.

Examples

>>> np.multiply.reduce([2,3,5])
30

A multi-dimensional array example:

>>> X = np.arange(8).reshape((2,2,2))
>>> X

```

```

array([[0, 1,
 [2, 3],
 [[4, 5],
 [6, 7]]])
>>> np.add.reduce(X, 0)
array([[4, 6],
 [8, 10]])
>>> np.add.reduce(X) # confirm: default axis value is 0
array([[4, 6],
 [8, 10]])
>>> np.add.reduce(X, 1)
array([[2, 4],
 [10, 12]])
>>> np.add.reduce(X, 2)
array([[1, 5],
 [9, 13]])

reduceat(...)
reduceat(a, indices, axis=0, dtype=None, out=None)

 Performs a (local) reduce with specified slices over a single axis.

 For i in ``range(len(indices))``, `reduceat` computes
 ``ufunc.reduce(a[indices[i]:indices[i+1]])``, which becomes the i-th
 generalized "row" parallel to `axis` in the final result (i.e., in a
 2-D array, for example, if `axis = 0`, it becomes the i-th row, but if
 `axis = 1`, it becomes the i-th column). There are three exceptions to this:

 * when ``i = len(indices) - 1`` (so for the last index),
 ``indices[i+1] = a.shape[axis]``.
 * if ``indices[i] >= indices[i + 1]``, the i-th generalized "row" is
 simply ``a[indices[i]]``.
 * if ``indices[i] >= len(a)`` or ``indices[i] < 0``, an error is raised.

 The shape of the output depends on the size of `indices`, and may be
 larger than `a` (this happens if ``len(indices) > a.shape[axis]``).

Parameters

a : array_like
 The array to act on.
indices : array_like
 Paired indices, comma separated (not colon), specifying slices to
 reduce.
axis : int, optional
 The axis along which to apply the reduceat.
dtype : data-type code, optional
 The type used to represent the intermediate results. Defaults
 to the data type of the output array if this is provided, or
 the data type of the input array if no output array is provided.
out : ndarray, optional
 A location into which the result is stored. If not provided a
 freshly-allocated array is returned.

Returns

r : ndarray
 The reduced values. If `out` was supplied, `r` is a reference to
 `out`.

Notes

A descriptive example:

If `a` is 1-D, the function `ufunc.accumulate(a)` is the same as
``ufunc.reduceat(a, indices)[::2]`` where `indices` is
``range(len(array) - 1)`` with a zero placed
in every other element:
``indices = zeros(2 * len(a) - 1)``, ``indices[1::2] = range(1, len(a))``.

Don't be fooled by this attribute's name: `reduceat(a)` is not
necessarily smaller than `a`.

Examples

To take the running sum of four successive values:

>>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]
array([6, 10, 14, 18])

A 2-D example:

>>> x = np.linspace(0, 15, 16).reshape(4,4)
>>> x
array([[0., 1., 2., 3.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.],
 [12., 13., 14., 15.]])

```

```
::
reduce such that the result has the following five rows:
[row1 + row2 + row3]
[row4]
[row2]
[row3]
[row1 + row2 + row3 + row4]

>>> np.add.reduceat(x, [0, 3, 1, 2, 0])
array([[12., 15., 18., 21.],
 [12., 13., 14., 15.],
 [4., 5., 6., 7.],
 [8., 9., 10., 11.],
 [24., 28., 32., 36.]])
::

reduce such that result has the following two columns:
[col1 * col2 * col3, col4]

>>> np.multiply.reduceat(x, [0, 3], 1)
array([[0., 3.],
 [120., 7.],
 [720., 11.],
 [2184., 15.]])
```

---

Data descriptors defined here:

### **identity**

The identity value.

Data attribute containing the identity element for the ufunc, if it has one.  
If it does not, the attribute value is None.

Examples

```

>>> np.add.identity
0
>>> np.multiply.identity
1
>>> np.power.identity
1
>>> print(np.exp.identity)
None
```

### **nargs**

The number of arguments.

Data attribute containing the number of arguments the ufunc takes, including optional ones.

Notes

Typically this value will be one more than what you might expect because all ufuncs take the optional "out" argument.

Examples

```

>>> np.add.nargs
3
>>> np.multiply.nargs
3
>>> np.power.nargs
3
>>> np.exp.nargs
2
```

### **nin**

The number of inputs.

Data attribute containing the number of arguments the ufunc treats as input.

Examples

```

>>> np.add.nin
2
>>> np.multiply.nin
2
>>> np.power.nin
2
>>> np.exp.nin
1
```

### **nout**

The number of outputs.

```
Data attribute containing the number of arguments the ufunc treats as output.
Notes

Since all ufuncs can take output arguments, this will always be (at least) 1.
Examples

>>> np.add.nout
1
>>> np.multiply.nout
1
>>> np.power.nout
1
>>> np.exp.nout
1
```

### **ntypes**

The number of types.

The number of numerical NumPy types - of which there are 18 total - on which the ufunc can operate.

See Also

[numpy.ufunc.types](#)

Examples

```

>>> np.add.ntypes
18
>>> np.multiply.ntypes
18
>>> np.power.ntypes
17
>>> np.exp.ntypes
7
>>> np.remainder.ntypes
14
```

### **signature**

#### **types**

Returns a list with types grouped input->output.

Data attribute listing the data-type "Domain-Range" groupings the ufunc can deliver. The data-types are given using the character codes.

See Also

[numpy.ufunc.ntypes](#)

Examples

```

>>> np.add.types
['?->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',<br/'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
'GG->G', 'OO->O']

>>> np.multiply.types
['?->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',<br/'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
'GG->G', 'OO->O']

>>> np.power.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
'OO->O']

>>> np.exp.types
['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']

>>> np.remainder.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']
```

**uint** = class uint64([unsignedinteger](#))

Method resolution order:

[uint64](#)  
[unsignedinteger](#)  
[integer](#)  
[number](#)

---

[generic](#)  
[builtin](#) [object](#)

---

Methods defined here:

---

[\\_\\_eq\\_\\_](#)(...)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_ge\\_\\_](#)(...)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_](#)(...)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_](#)(...)  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_index\\_\\_](#)(...)  
x[y:z] <==> x[y. [\\_\\_index\\_\\_](#)():z. [\\_\\_index\\_\\_](#)()]

[\\_\\_le\\_\\_](#)(...)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_lt\\_\\_](#)(...)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_](#)(...)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

---

Data and other attributes defined here:

---

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

---

[\\_\\_abs\\_\\_](#)(...)  
x. [\\_\\_abs\\_\\_](#)() <==> abs(x)

[\\_\\_add\\_\\_](#)(...)  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y

[\\_\\_and\\_\\_](#)(...)  
x. [\\_\\_and\\_\\_](#)(y) <==> x&y

[\\_\\_array\\_\\_](#)(...)  
sc. [\\_\\_array\\_\\_](#)(|type) return 0-dim array

[\\_\\_array\\_wrap\\_\\_](#)(...)  
sc. [\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array

[\\_\\_copy\\_\\_](#)(...)

[\\_\\_deepcopy\\_\\_](#)(...)

[\\_\\_div\\_\\_](#)(...)  
x. [\\_\\_div\\_\\_](#)(y) <==> x/y

[\\_\\_divmod\\_\\_](#)(...)  
x. [\\_\\_divmod\\_\\_](#)(y) <==> divmod(x, y)

[\\_\\_float\\_\\_](#)(...)  
x. [\\_\\_float\\_\\_](#)() <==> [float](#)(x)

```
floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

repr(...)
x._repr_() <==> repr(x)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

rmod(...)
x._rmod_(y) <==> y%x
```

**\_\_rmul\_\_**(...)  
x. \_\_rmul\_\_(y) <==> y\*x

**\_\_ror\_\_**(...)  
x. \_\_ror\_\_(y) <==> y|x

**\_\_rpow\_\_**(...)  
y. \_\_rpow\_\_(x[, z]) <==> pow(x, y[, z])

**\_\_rrshift\_\_**(...)  
x. \_\_rrshift\_\_(y) <==> y>>x

**\_\_rshift\_\_**(...)  
x. \_\_rshift\_\_(y) <==> x>>y

**\_\_rsub\_\_**(...)  
x. \_\_rsub\_\_(y) <==> y-x

**\_\_rtruediv\_\_**(...)  
x. \_\_rtruediv\_\_(y) <==> y/x

**\_\_rxor\_\_**(...)  
x. \_\_rxor\_\_(y) <==> y^x

**\_\_setstate\_\_**(...)

**\_\_sizeof\_\_**(...)

**\_\_str\_\_**(...)  
x. \_\_str\_\_() <==> str(x)

**\_\_sub\_\_**(...)  
x. \_\_sub\_\_(y) <==> x-y

**\_\_truediv\_\_**(...)  
x. \_\_truediv\_\_(y) <==> x/y

**\_\_xor\_\_**(...)  
x. \_\_xor\_\_(y) <==> x^y

**all(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**any(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin(...)**  
Not implemented (virtual attribute)  
  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

```

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of 'new_order' for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape\(...\)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

**uint0** = class uint64([unsignedinteger](#))

Method resolution order:

[uint64](#)  
[unsignedinteger](#)  
[integer](#)  
[number](#)  
[generic](#)  
[builtin](#) .[object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_\(...\)](#)  
x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

[\\_\\_ge\\_\\_\(...\)](#)  
x. [\\_\\_ge\\_\\_](#)(y) <==> x>=y

[\\_\\_gt\\_\\_\(...\)](#)  
x. [\\_\\_gt\\_\\_](#)(y) <==> x>y

[\\_\\_hash\\_\\_\(...\)](#)  
x. [\\_\\_hash\\_\\_](#)() <==> hash(x)

[\\_\\_index\\_\\_\(...\)](#)  
x[y:z] <==> x[y. [\\_\\_index\\_\\_](#)():z. [\\_\\_index\\_\\_](#)()]

[\\_\\_le\\_\\_\(...\)](#)  
x. [\\_\\_le\\_\\_](#)(y) <==> x<=y

[\\_\\_lt\\_\\_\(...\)](#)  
x. [\\_\\_lt\\_\\_](#)(y) <==> x<y

[\\_\\_ne\\_\\_\(...\)](#)  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

[\\_\\_abs\\_\\_\(...\)](#)  
x. [\\_\\_abs\\_\\_](#)() <==> abs(x)

[\\_\\_add\\_\\_\(...\)](#)  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y

[\\_\\_and\\_\\_\(...\)](#)

```
x. and (y) <==> x&y
array(...) sc. array (|type) return 0-dim array
array_wrap(...) sc. array_wrap (obj) return scalar from array
copy(...)
deepcopy(...)
div(...) x. div (y) <==> x/y
divmod(...) x. divmod (y) <==> divmod(x, y)
float(...) x. float () <==> float(x)
floordiv(...) x. floordiv (y) <==> x//y
format(...) NumPy array scalar formatter
getitem(...) x. getitem (y) <==> x[y]
hex(...) x. hex () <==> hex(x)
int(...) x. int () <==> int(x)
invert(...) x. invert () <==> ~x
long(...) x. long () <==> long(x)
lshift(...) x. lshift (y) <==> x<<y
mod(...) x. mod (y) <==> x%y
mul(...) x. mul (y) <==> x*y
neg(...) x. neg () <==> -x
nonzero(...) x. nonzero () <==> x != 0
oct(...) x. oct () <==> oct(x)
or(...) x. or (y) <==> x|y
pos(...) x. pos () <==> +x
pow(...) x. pow (y[, z]) <==> pow(x, y[, z])
radd(...) x. radd (y) <==> y+x
rand(...) x. rand (y) <==> y&x
```

```
__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmax(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

[new\\_order](#) : [str](#), optional

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

[new\\_dtype](#) : [dtype](#)

New `dtype` object with the given change to the [byte](#) order.

### **nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **put(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **repeat(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **reshape(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

### class [uint16](#)([unsignedinteger](#))

Method resolution order:

```
uint16
unsignedinteger
integer
number
generic
builtin object
```

---

Methods defined here:

```
__eq__\(...\)
x. __eq__\(y\) <==> x==y

__ge__\(...\)
x. __ge__\(y\) <==> x>=y

__gt__\(...\)
x. __gt__\(y\) <==> x>y

__hash__\(...\)
x. __hash__\(\) <==> hash(x)

__index__\(...\)
x[y:z] <==> x[y. __index__\(\):z. __index__\(\)]

__le__\(...\)
x. __le__\(y\) <==> x<=y

__lt__\(...\)
x. __lt__\(y\) <==> x<y

__ne__\(...\)
x. __ne__\(y\) <==> x!=y
```

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T.**\_\_new\_\_**(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x.**\_\_abs\_\_**() <==> abs(x)

**\_\_add\_\_(...)**  
x.**\_\_add\_\_**(y) <==> x+y

**\_\_and\_\_(...)**  
x.**\_\_and\_\_**(y) <==> x&y

**\_\_array\_\_(...)**  
sc.**\_\_array\_\_**(|type) return 0-dim array

**\_\_array\_wrap\_\_(...)**  
sc.**\_\_array\_wrap\_\_**(obj) return scalar from array

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**  
x.**\_\_div\_\_**(y) <==> x/y

**\_\_divmod\_\_(...)**  
x.**\_\_divmod\_\_**(y) <==> divmod(x, y)

**\_\_float\_\_(...)**  
x.**\_\_float\_\_**() <==> float(x)

**\_\_floordiv\_\_(...)**  
x.**\_\_floordiv\_\_**(y) <==> x//y

**\_\_format\_\_(...)**  
NumPy array scalar formatter

**\_\_getitem\_\_(...)**  
x.**\_\_getitem\_\_**(y) <==> x[y]

**\_\_hex\_\_(...)**  
x.**\_\_hex\_\_**() <==> hex(x)

**\_\_int\_\_(...)**  
x.**\_\_int\_\_**() <==> int(x)

**\_\_invert\_\_(...)**  
x.**\_\_invert\_\_**() <==> ~x

**\_\_long\_\_(...)**  
x.**\_\_long\_\_**() <==> long(x)

**\_\_lshift\_\_(...)**  
x.**\_\_lshift\_\_**(y) <==> x<<y

**\_\_mod\_\_(...)**  
x.**\_\_mod\_\_**(y) <==> x%y

**\_\_mul\_\_(...)**  
x.**\_\_mul\_\_**(y) <==> x\*y

**\_\_neg\_\_(...)**  
x.**\_\_neg\_\_**() <==> -x

```
nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%x

rmul(...)
x.rmul(y) <==> y*x

ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y
```

```
truediv(...)
x._truediv_(y) <==> x/y

xor(...)
x._xor_(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

byteswap(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class so as to
provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New [dtype](#) object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
tobytes(...)
tofile(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tolist(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

toString(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

```
T
 transpose

_array_interface_
 Array protocol: Python side

_array_priority_
 Array priority.

_array_struct_
 Array protocol: struct

base
 base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

class uint32(unsignedinteger)
Method resolution order:
 uint32
 unsignedinteger
 integer
 number
 generic
 builtin object
```

---

Methods defined here:

```
eq(...)
 x.eq(y) <==> x==y

ge(...)
 x.ge(y) <==> x>=y

gt(...)
 x.gt(y) <==> x>y
```

---

```

hash(...)
 x._hash_() <==> hash(x)

index(...)
 x[y:z] <==> x[y._index_():z._index_()]

le(...)
 x._le_(y) <==> x<=y

lt(...)
 x._lt_(y) <==> x<y

ne(...)
 x._ne_(y) <==> x!=y

```

---

Data and other attributes defined here:

---

```

new = <built-in method _new_ of type object>
 T._new_(S, ...) -> a new object with type S, a subtype of T

```

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

---

```

abs(...)
 x._abs_() <==> abs(x)

add(...)
 x._add_(y) <==> x+y

and(...)
 x._and_(y) <==> x&y

array(...)
 sc._array_(|type) return 0-dim array

_array_wrap_(...)
 sc._array_wrap_(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
 x._div_(y) <==> x/y

divmod(...)
 x._divmod_(y) <==> divmod(x, y)

float(...)
 x._float_() <==> float(x)

floordiv(...)
 x._floordiv_(y) <==> x//y

format(...)
 NumPy array scalar formatter

getitem(...)
 x._getitem_(y) <==> x[y]

hex(...)
 x._hex_() <==> hex(x)

int(...)
 x._int_() <==> int(x)

```

```
__invert__(...)
x.__invert__() <==> ~x

__long__(...)
x.__long__() <==> long(x)

__lshift__(...)
x.__lshift__(y) <==> x<<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%y

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y
```

```
__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

astype(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

- \* 'S' - swap [dtype](#) from current to opposite endian
- \* {'<', 'L'} - little endian
- \* {'>', 'B'} - big endian
- \* {'=', 'N'} - native order
- \* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

[new\\_order](#) : [str](#), optional

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

[new\\_dtype](#) : [dtype](#)

New `dtype` [object](#) with the given change to the [byte](#) order.

### **nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**put(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**repeat(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**reshape(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
swapaxes(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

take(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tobytes(...)
tofile(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

tolist(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

toString(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_\_array\_interface\_\_**  
Array protocol: Python side

**\_\_array\_priority\_\_**  
Array priority.

**\_\_array\_struct\_\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

```
class uint64(unsignedinteger)
 Method resolution order:
 uint64
 unsignedinteger
 integer
 number
 generic
 builtin _object
```

---

Methods defined here:

```
__eq__(...)
x. __eq__(y) <==> x==y

__ge__(...)
x. __ge__(y) <==> x>=y

__gt__(...)
x. __gt__(y) <==> x>y

__hash__(...)
x. __hash__() <==> hash(x)

__index__(...)
x[y:z] <==> x[y. __index__():z. __index__()]

__le__(...)
x. __le__(y) <==> x<=y

__lt__(...)
x. __lt__(y) <==> x<y

__ne__(...)
x. __ne__(y) <==> x!=y
```

---

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T. __new__(S, ...) -> a new object with type S, a subtype of T
```

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

```
__abs__(...)
x. __abs__() <==> abs(x)

__add__(...)
x. __add__(y) <==> x+y

__and__(...)
x. __and__(y) <==> x&y

__array__(...)
sc. __array__(|type) return 0-dim array

__array_wrap__(...)
sc. __array_wrap__(obj) return scalar from array

__copy__(...)

__deepcopy__(...)

__div__(...)
x. __div__(y) <==> x/y
```

```
divmod(...)
x._divmod_(y) <==> divmod(x, y)

float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x._getitem_(y) <==> x[y]

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

mod(...)
x._mod_(y) <==> x%y

mul(...)
x._mul_(y) <==> x*y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

repr(...)
x._repr_() <==> repr(x)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x
```

```
__rlshift__(...)
 x.__rlshift__(y) <==> y<<x

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(y) <==> y*x

__ror__(...)
 x.__ror__(y) <==> y|x

__rpow__(...)
 y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x.__rrshift__(y) <==> y>>x

__rshift__(...)
 x.__rshift__(y) <==> x>>y

__rsub__(...)
 x.__rsub__(y) <==> y-x

__rtruediv__(...)
 x.__rtruediv__(y) <==> y/x

__rxor__(...)
 x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x.__str__() <==> str(x)

__sub__(...)
 x.__sub__(y) <==> x-y

__truediv__(...)
 x.__truediv__(y) <==> x/y

__xor__(...)
 x.__xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**argmin(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:
```

```
* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**reshape(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

std(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tostring(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

```
trace(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

transpose(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

var(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

view(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

---

class **uint8**([unsignedinteger](#))

Method resolution order:

```

uint8
 unsignedinteger
 integer
 number
 generic
 builtin object

```

---

Methods defined here:

```

eq(...)
 x. _eq_(y) <==> x==y

ge(...)
 x. _ge_(y) <==> x>=y

gt(...)
 x. _gt_(y) <==> x>y

hash(...)
 x. _hash_() <==> hash(x)

index(...)
 x[y:z] <==> x[y. _index_():z. _index_()]

le(...)
 x. _le_(y) <==> x<=y

lt(...)
 x. _lt_(y) <==> x<y

ne(...)
 x. _ne_(y) <==> x!=y

```

---

Data and other attributes defined here:

```

new = <built-in method _new_ of type object>
 T. _new_(S, ...) -> a new object with type S, a subtype of T

```

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_abs\_**(...)

```
x.abs() <==> abs(x)

add(...)
x.add(y) <==> x+y

and(...)
x.and(y) <==> x&y

array(...)
sc.array(|type) return 0-dim array

array_wrap(...)
sc.array_wrap(obj) return scalar from array

copy(...)

deepcopy(...)

div(...)
x.div(y) <==> x/y

divmod(...)
x.divmod(y) <==> divmod(x, y)

float(...)
x.float() <==> float(x)

floordiv(...)
x.floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x.getitem(y) <==> x[y]

hex(...)
x.hex() <==> hex(x)

int(...)
x.int() <==> int(x)

invert(...)
x.invert() <==> ~x

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

neg(...)
x.neg() <==> -x

nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])
```

```
__radd__(...)
x. __radd__(y) <==> y+x

__rand__(...)
x. __rand__(y) <==> y&x

__rdiv__(...)
x. __rdiv__(y) <==> y/x

__rdivmod__(...)
x. __rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x. __repr__() <==> repr(x)

__rfloordiv__(...)
x. __rfloordiv__(y) <==> y//x

__rlshift__(...)
x. __rlshift__(y) <==> y<<x

__rmod__(...)
x. __rmod__(y) <==> y%x

__rmul__(...)
x. __rmul__(y) <==> y*x

__ror__(...)
x. __ror__(y) <==> y|x

__rpow__(...)
y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x. __rrshift__(y) <==> y>>x

__rshift__(...)
x. __rshift__(y) <==> x>>y

__rsub__(...)
x. __rsub__(y) <==> y-x

__rtruediv__(...)
x. __rtruediv__(y) <==> y/x

__rxor__(...)
x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
x. __str__() <==> str(x)

__sub__(...)
x. __sub__(y) <==> x-y

__truediv__(...)
x. __truediv__(y) <==> x/y

__xor__(...)
x. __xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**any(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmax(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.
min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to

provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**transpose(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)  
  
Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.  
  
See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**

```
base object

data
 pointer to start of data

dtype
 get array data-descriptor

flags
 integer value of flags

flat
 a 1-d view of scalar

imag
 imaginary part of scalar

itemsize
 length of one element in bytes

nbytes
 length of item in bytes

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension
```

---

**uintc** = class uint32([unsignedinteger](#))

Method resolution order:

```
uint32
unsignedinteger
integer
number
generic
builtin object
```

---

Methods defined here:

```
__eq__(...)
 x. __eq__(y) <==> x==y

__ge__(...)
 x. __ge__(y) <==> x>=y

__gt__(...)
 x. __gt__(y) <==> x>y

__hash__(...)
 x. __hash__() <==> hash(x)

__index__(...)
 x[y:z] <==> x[y. __index__():z. __index__()]

__le__(...)
 x. __le__(y) <==> x<=y

__lt__(...)
 x. __lt__(y) <==> x<y
```

**\_\_ne\_\_(...)**  
x. [\\_\\_ne\\_\\_](#)(y) <==> x!=y

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x. [\\_\\_abs\\_\\_](#)() <==> abs(x)

**\_\_add\_\_(...)**  
x. [\\_\\_add\\_\\_](#)(y) <==> x+y

**\_\_and\_\_(...)**  
x. [\\_\\_and\\_\\_](#)(y) <==> x&y

**\_\_array\_\_(...)**  
sc. [\\_\\_array\\_\\_](#)(|type) return 0-dim array

**\_\_array\_wrap\_\_(...)**  
sc. [\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**  
x. [\\_\\_div\\_\\_](#)(y) <==> x/y

**\_\_divmod\_\_(...)**  
x. [\\_\\_divmod\\_\\_](#)(y) <==> divmod(x, y)

**\_\_float\_\_(...)**  
x. [\\_\\_float\\_\\_](#)() <==> float(x)

**\_\_floordiv\_\_(...)**  
x. [\\_\\_floordiv\\_\\_](#)(y) <==> x//y

**\_\_format\_\_(...)**  
NumPy array scalar formatter

**\_\_getitem\_\_(...)**  
x. [\\_\\_getitem\\_\\_](#)(y) <==> x[y]

**\_\_hex\_\_(...)**  
x. [\\_\\_hex\\_\\_](#)() <==> hex(x)

**\_\_int\_\_(...)**  
x. [\\_\\_int\\_\\_](#)() <==> int(x)

**\_\_invert\_\_(...)**  
x. [\\_\\_invert\\_\\_](#)() <==> ~x

**\_\_long\_\_(...)**  
x. [\\_\\_long\\_\\_](#)() <==> long(x)

**\_\_lshift\_\_(...)**  
x. [\\_\\_lshift\\_\\_](#)(y) <==> x<<y

**\_\_mod\_\_(...)**  
x. [\\_\\_mod\\_\\_](#)(y) <==> x%y

```
__mul__(...)
x.__mul__(y) <==> x*y

__neg__(...)
x.__neg__() <==> -x

__nonzero__(...)
x.__nonzero__() <==> x != 0

__oct__(...)
x.__oct__() <==> oct(x)

__or__(...)
x.__or__(y) <==> x|y

__pos__(...)
x.__pos__() <==> +x

__pow__(...)
x.__pow__(y[, z]) <==> pow(x, y[, z])

__radd__(...)
x.__radd__(y) <==> y+x

__rand__(...)
x.__rand__(y) <==> y&x

__rdiv__(...)
x.__rdiv__(y) <==> y/x

__rdivmod__(...)
x.__rdivmod__(y) <==> divmod(y, x)

__reduce__(...)

__repr__(...)
x.__repr__() <==> repr(x)

__rfloordiv__(...)
x.__rfloordiv__(y) <==> y//x

__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)
```

**\_\_str\_\_**(...)  
x.\_\_str\_\_() <==> str(x)

**\_\_sub\_\_**(...)  
x.\_\_sub\_\_(y) <==> x-y

**\_\_truediv\_\_**(...)  
x.\_\_truediv\_\_(y) <==> x/y

**\_\_xor\_\_**(...)  
x.\_\_xor\_\_(y) <==> x^y

**all**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**any**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmax**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argmin**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap**(...)  
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to

provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**ptp(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**put(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_array\_interface\_**  
    Array protocol: Python side

**\_array\_priority\_**  
    Array priority.

**\_array\_struct\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

**uintp** = class uint64([unsignedinteger](#))

Method resolution order:

[uint64](#)  
[unsignedinteger](#)  
[integer](#)  
[number](#)  
[generic](#)  
[builtin\\_object](#)

---

Methods defined here:

**\_eq\_**(...)  
x. [\\_eq\\_](#)(y) <==> x==y

---

**\_\_ge\_\_**(...)  
 $x.\underline{\text{ge}}(y) \iff x \geq y$   
**\_\_gt\_\_**(...)  
 $x.\underline{\text{gt}}(y) \iff x > y$   
**\_\_hash\_\_**(...)  
 $x.\underline{\text{hash}}() \iff \text{hash}(x)$   
**\_\_index\_\_**(...)  
 $x[y:z] \iff x[y.\underline{\text{index}}():z.\underline{\text{index}}()]$   
**\_\_le\_\_**(...)  
 $x.\underline{\text{le}}(y) \iff x \leq y$   
**\_\_lt\_\_**(...)  
 $x.\underline{\text{lt}}(y) \iff x < y$   
**\_\_ne\_\_**(...)  
 $x.\underline{\text{ne}}(y) \iff x \neq y$

---

Data and other attributes defined here:

---

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
 $T.\underline{\text{new}}(S, \dots) \rightarrow$  a new object with type S, a subtype of T

---

Data descriptors inherited from integer:

**denominator**  
 $\text{denominator}$  of value (1)

**numerator**  
 $\text{numerator}$  of value (the value itself)

---

Methods inherited from generic:

---

**\_\_abs\_\_**(...)  
 $x.\underline{\text{abs}}() \iff \text{abs}(x)$   
**\_\_add\_\_**(...)  
 $x.\underline{\text{add}}(y) \iff x+y$   
**\_\_and\_\_**(...)  
 $x.\underline{\text{and}}(y) \iff x\&y$   
**\_\_array\_\_**(...)  
 $\text{sc.}\underline{\text{array}}(|\text{type}) \text{ return 0-dim array}$   
**\_\_array\_wrap\_\_**(...)  
 $\text{sc.}\underline{\text{array wrap}}(\text{obj}) \text{ return scalar from array}$   
**\_\_copy\_\_**(...)  
**\_\_deepcopy\_\_**(...)  
**\_\_div\_\_**(...)  
 $x.\underline{\text{div}}(y) \iff x/y$   
**\_\_divmod\_\_**(...)  
 $x.\underline{\text{divmod}}(y) \iff \text{divmod}(x, y)$   
**\_\_float\_\_**(...)  
 $x.\underline{\text{float}}() \iff \text{float}(x)$   
**\_\_floordiv\_\_**(...)  
 $x.\underline{\text{floordiv}}(y) \iff x//y$   
**\_\_format\_\_**(...)  
 $\text{NumPy array scalar formatter}$   
**\_\_getitem\_\_**(...)  
 $x.\underline{\text{getitem}}(y) \iff x[y]$

```
hex(...)
x.hex() <==> hex(x)
int(...)
x.int() <==> int(x)
invert(...)
x.invert() <==> ~x
long(...)
x.long() <==> long(x)
lshift(...)
x.lshift(y) <==> x<<y
mod(...)
x.mod(y) <==> x%y
mul(...)
x.mul(y) <==> x*y
neg(...)
x.neg() <==> -x
nonzero(...)
x.nonzero() <==> x != 0
oct(...)
x.oct() <==> oct(x)
or(...)
x.or(y) <==> x|y
pos(...)
x.pos() <==> +x
pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])
radd(...)
x.radd(y) <==> y+x
rand(...)
x.rand(y) <==> y&x
rdiv(...)
x.rdiv(y) <==> y/x
rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)
reduce(...)
repr(...)
x.repr() <==> repr(x)
rfloordiv(...)
x.rfloordiv(y) <==> y//x
rlshift(...)
x.rlshift(y) <==> y<<x
rmod(...)
x.rmod(y) <==> y%x
rmul(...)
x.rmul(y) <==> y*x
ror(...)
x.ror(y) <==> y|x
rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])
```

```
__rrshift__(...)
 x. __rrshift__(y) <==> y>>x

__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
```

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
```

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

```
See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

**ulonglong = class uint64([unsignedinteger](#))**

Method resolution order:

```
 uint64
 unsignedinteger
 integer
 number
 generic
 builtin__object
```

Methods defined here:

```
__eq__\(...\)
x. __eq__\(y\) <==> x==y

__ge__\(...\)
x. __ge__\(y\) <==> x>=y

__gt__\(...\)
x. __gt__\(y\) <==> x>y

__hash__\(...\)
x. __hash__\(\) <==> hash(x)

__index__\(...\)
x[y:z] <==> x[y. __index__\(\):z. __index__\(\)]

__le__\(...\)
x. __le__\(y\) <==> x<=y

__lt__\(...\)
x. __lt__\(y\) <==> x<y

__ne__\(...\)
x. __ne__\(y\) <==> x!=y
```

Data and other attributes defined here:

```
__new__ = <built-in method __new__ of type object>
T. __new__\(S, ...\) -> a new object with type S, a subtype of T
```

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

Methods inherited from [generic](#):

```
__abs__\(...\)
x. __abs__\(\) <==> abs(x)

__add__\(...\)
x. __add__\(y\) <==> x+y

__and__\(...\)
x. __and__\(y\) <==> x&y

__array__\(...\)
sc. __array__\(\)|type) return 0-dim array

__array_wrap__\(...\)
sc. __array_wrap__\(obj\) return scalar from array

__copy__\(...\)
```

```
__deepcopy__(...)
__div__(...)
 x.__div__(y) <==> x/y
__divmod__(...)
 x.__divmod__(y) <==> divmod(x, y)
__float__(...)
 x.__float__() <==> float(x)
__floordiv__(...)
 x.__floordiv__(y) <==> x//y
__format__(...)
 NumPy array scalar formatter
__getitem__(...)
 x.__getitem__(y) <==> x[y]
__hex__(...)
 x.__hex__() <==> hex(x)
__int__(...)
 x.__int__() <==> int(x)
__invert__(...)
 x.__invert__() <==> ~x
__long__(...)
 x.__long__() <==> long(x)
__lshift__(...)
 x.__lshift__(y) <==> x<<y
__mod__(...)
 x.__mod__(y) <==> x%y
__mul__(...)
 x.__mul__(y) <==> x*y
__neg__(...)
 x.__neg__() <==> -x
__nonzero__(...)
 x.__nonzero__() <==> x != 0
__oct__(...)
 x.__oct__() <==> oct(x)
__or__(...)
 x.__or__(y) <==> x|y
__pos__(...)
 x.__pos__() <==> +x
__pow__(...)
 x.__pow__(y[, z]) <==> pow(x, y[, z])
__radd__(...)
 x.__radd__(y) <==> y+x
__rand__(...)
 x.__rand__(y) <==> y&x
__rdiv__(...)
 x.__rdiv__(y) <==> y/x
__rdivmod__(...)
 x.__rdivmod__(y) <==> divmod(y, x)
__reduce__(...)
__repr__(...)
```

```
x.__repr__() <==> repr(x)

_rfloordiv__(...)
x._rfloordiv__(y) <==> y//x

_rlshift__(...)
x._rlshift__(y) <==> y<<x

_rmod__(...)
x._rmod__(y) <==> y%x

_rmul__(...)
x._rmul__(y) <==> y*x

_ror__(...)
x._ror__(y) <==> y|x

_rpow__(...)
y._rpow__(x[, z]) <==> pow(x, y[, z])

_rrshift__(...)
x._rrshift__(y) <==> y>>x

_rshift__(...)
x._rshift__(y) <==> x>>y

_rsub__(...)
x._rsub__(y) <==> y-x

_rtruediv__(...)
x._rtruediv__(y) <==> y/x

_rxor__(...)
x._rxor__(y) <==> y^x

_setstate__(...)

_sizeof__(...)

_str__(...)
x._str__() <==> str(x)

_sub__(...)
x._sub__(y) <==> x-y

_truediv__(...)
x._truediv__(y) <==> x/y

_xor__(...)
x._xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
```

albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

### **argmin(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **argsort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **astype(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **byteswap(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **choose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **clip(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **compress(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**conj(...)****conjugate(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

```
newbyteorder(new_order='S')

 Return a new `dtype` with a different byte order.

 Changes are also made in all fields and sub-arrays of the data type.

 The `new_order` code can be any from the following:

 * 'S' - swap dtype from current to opposite endian
 * {'<', 'L'} - little endian
 * {'>', 'B'} - big endian
 * {'=', 'N'} - native order
 * {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

prod(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ptp(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

put(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

ravel(...)

 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

repeat(...)

 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**toString(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

```
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    **transpose**

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

---

**unicode0** = class `unicode_( builtin_unicode, character)`

Method resolution order:

[unicode](#)  
[builtin\\_unicode](#)  
[builtin\\_basestring](#)  
[character](#)  
[flexible](#)  
[generic](#)  
[builtin\\_object](#)

---

Methods defined here:

[\\_\\_eq\\_\\_\(...\)](#)  
`x. __eq__(y) <==> x==y`

[\\_\\_ge\\_\\_\(...\)](#)  
`x. __ge__(y) <==> x>=y`

[\\_\\_gt\\_\\_\(...\)](#)  
`x. __gt__(y) <==> x>y`

[\\_\\_hash\\_\\_\(...\)](#)  
`x. __hash__() <==> hash(x)`

[\\_\\_le\\_\\_\(...\)](#)  
`x. __le__(y) <==> x<=y`

[\\_\\_lt\\_\\_\(...\)](#)  
`x. __lt__(y) <==> x<y`

[\\_\\_ne\\_\\_\(...\)](#)  
`x. __ne__(y) <==> x!=y`

[\\_\\_repr\\_\\_\(...\)](#)  
`x. __repr__() <==> repr(x)`

[\\_\\_str\\_\\_\(...\)](#)  
`x. __str__() <==> str(x)`

---

Data and other attributes defined here:

[\\_\\_new\\_\\_](#) = <built-in method `__new__` of type object>  
`T. __new__(S, ...) -> a new object with type S, a subtype of T`

---

Methods inherited from [builtin\\_unicode](#):

```
__add__(...)
 x.__add__(y) <==> x+y

__contains__(...)
 x.__contains__(y) <==> y in x

__format__(...)
 S.__format__(format_spec) -> unicode

 Return a formatted version of S as described by format_spec.

__getattribute__(...)
 x.__getattribute__('name') <==> x.name

__getitem__(...)
 x.__getitem__(y) <==> x[y]

__getnewargs__(...)

__getslice__(...)
 x.__getslice__(i, j) <==> x[i:j]

 Use of negative indices is not supported.

__len__(...)
 x.__len__() <==> len(x)

__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(n) <==> x*n

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(n) <==> n*x

__sizeof__(...)
 S.__sizeof__() -> size of S in memory, in bytes

capitalize(...)
 S.capitalize() -> unicode

 Return a capitalized version of S, i.e. make the first character
 have upper case and the rest lower case.

center(...)
 S.center(width[, fillchar]) -> unicode

 Return S centered in a Unicode string of length width. Padding is
 done using the specified fill character (default is a space)

count(...)
 S.count(sub[, start[, end]]) -> int

 Return the number of non-overlapping occurrences of substring sub in
 Unicode string S[start:end]. Optional arguments start and end are
 interpreted as in slice notation.

decode(...)
 S.decode([encoding[,errors]]) -> string or unicode

 Decodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'
 as well as any other name registered with codecs.register_error that is
 able to handle UnicodeDecodeErrors.

encode(...)
 S.encode([encoding[,errors]]) -> string or unicode

 Encodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
 'xmlcharrefreplace' as well as any other name registered with
```

```
codecs.register_error that can handle UnicodeEncodeErrors.

endswith(...)
S.endswith(suffix[, start[, end]]) -> bool
 Return True if S ends with the specified suffix, False otherwise.
 With optional start, test S beginning at that position.
 With optional end, stop comparing S at that position.
 suffix can also be a tuple of strings to try.

expandtabs(...)
S.expandtabs([tabsize]) -> unicode
 Return a copy of S where all tab characters are expanded using spaces.
 If tabsize is not given, a tab size of 8 characters is assumed.

find(...)
S.find(sub [,start [,end]]) -> int
 Return the lowest index in S where substring sub is found,
 such that sub is contained within S[start:end]. Optional
 arguments start and end are interpreted as in slice notation.

 Return -1 on failure.

format(...)
S.format(*args, **kwargs) -> unicode
 Return a formatted version of S, using substitutions from args and kwargs.
 The substitutions are identified by braces ('{' and '}').

index(...)
S.index(sub [,start [,end]]) -> int
 Like S.find() but raise ValueError when the substring is not found.

isalnum(...)
S.isalnum() -> bool
 Return True if all characters in S are alphanumeric
 and there is at least one character in S, False otherwise.

isalpha(...)
S.isalpha() -> bool
 Return True if all characters in S are alphabetic
 and there is at least one character in S, False otherwise.

isdecimal(...)
S.isdecimal() -> bool
 Return True if there are only decimal characters in S,
 False otherwise.

isdigit(...)
S.isdigit() -> bool
 Return True if all characters in S are digits
 and there is at least one character in S, False otherwise.

islower(...)
S.islower() -> bool
 Return True if all cased characters in S are lowercase and there is
 at least one cased character in S, False otherwise.

isnumeric(...)
S.isnumeric() -> bool
 Return True if there are only numeric characters in S,
 False otherwise.

isspace(...)
S.isspace() -> bool
 Return True if all characters in S are whitespace
 and there is at least one character in S, False otherwise.

istitle(...)
S.istitle() -> bool
```

Return True if S is a titlecased string and there is at least one [character](#) in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones.

Return False otherwise.

**isupper(...)**  
S.[isupper\(\)](#) -> bool

Return True if all cased characters in S are uppercase and there is at least one cased [character](#) in S, False otherwise.

**join(...)**  
S.[join](#)(iterable) -> [unicode](#)

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

**ljust(...)**  
S.[ljust](#)(width[, fillchar]) -> [int](#)

Return S left-justified in a Unicode string of length width. Padding is done using the specified fill [character](#) (default is a space).

**lower(...)**  
S.[lower\(\)](#) -> [unicode](#)

Return a copy of the string S converted to lowercase.

**lstrip(...)**  
S.[lstrip](#)([chars]) -> [unicode](#)

Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a [str](#), it will be converted to [unicode](#) before stripping

**partition(...)**  
S.[partition](#)(sep) -> (head, sep, tail)

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

**replace(...)**  
S.[replace](#)(old, new[, count]) -> [unicode](#)

Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

**rfind(...)**  
S.[rfind](#)(sub [,start [,end]]) -> [int](#)

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

**rindex(...)**  
S.[rindex](#)(sub [,start [,end]]) -> [int](#)

Like S.[rfind\(\)](#) but raise ValueError when the substring is not found.

**rjust(...)**  
S.[rjust](#)(width[, fillchar]) -> [unicode](#)

Return S right-justified in a Unicode string of length width. Padding is done using the specified fill [character](#) (default is a space).

**rpartition(...)**  
S.[rpartition](#)(sep) -> (head, sep, tail)

Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

**rsplit(...)**  
S.[rsplit](#)([sep [,maxsplit]]) -> list of strings

Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit

splits are done. If sep is not specified, any whitespace string is a separator.

**rstrip(...)**  
S.[rstrip](#)([chars]) -> [unicode](#)  
Return a copy of the string S with trailing whitespace removed.  
If chars is given and not None, remove characters in chars instead.  
If chars is a [str](#), it will be converted to [unicode](#) before stripping

**split(...)**  
S.[split](#)([sep [,maxsplit]]) -> list of strings  
Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

**splitlines(...)**  
S.[splitlines](#)(keepends=False) -> list of strings  
Return a list of the lines in S, breaking at line boundaries.  
Line breaks are not included in the resulting list unless keepends is given and true.

**startswith(...)**  
S.[startswith](#)(prefix[, start[, end]]) -> bool  
Return True if S starts with the specified prefix, False otherwise.  
With optional start, test S beginning at that position.  
With optional end, stop comparing S at that position.  
prefix can also be a tuple of strings to try.

**strip(...)**  
S.[strip](#)([chars]) -> [unicode](#)  
Return a copy of the string S with leading and trailing whitespace removed.  
If chars is given and not None, remove characters in chars instead.  
If chars is a [str](#), it will be converted to [unicode](#) before stripping

**swapcase(...)**  
S.[swapcase](#)() -> [unicode](#)  
Return a copy of S with uppercase characters converted to lowercase and vice versa.

**title(...)**  
S.[title](#)() -> [unicode](#)  
Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

**translate(...)**  
S.[translate](#)(table) -> [unicode](#)  
Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None. Unmapped characters are left untouched. Characters mapped to None are deleted.

**upper(...)**  
S.[upper](#)() -> [unicode](#)  
Return a copy of S converted to uppercase.

**zfill(...)**  
S.[zfill](#)(width) -> [unicode](#)  
Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

---

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
x.[\\_\\_abs\\_\\_](#)() <=> abs(x)

**\_\_and\_\_(...)**

```
x. and_(y) <==> x&y
array_(...)
 sc. array_(|type) return 0-dim array
array_wrap_(...)
 sc. array_wrap_(obj) return scalar from array
copy_(...)
deepcopy_(...)
div_(...)
 x. div_(y) <==> x/y
divmod_(...)
 x. divmod_(y) <==> divmod(x, y)
float_(...)
 x. float_() <==> float(x)
floordiv_(...)
 x. floordiv_(y) <==> x//y
hex_(...)
 x. hex_() <==> hex(x)
int_(...)
 x. int_() <==> int(x)
invert_(...)
 x. invert_() <==> ~x
long_(...)
 x. long_() <==> long(x)
lshift_(...)
 x. lshift_(y) <==> x<<y
neg_(...)
 x. neg_() <==> -x
nonzero_(...)
 x. nonzero_() <==> x != 0
oct_(...)
 x. oct_() <==> oct(x)
or_(...)
 x. or_(y) <==> x|y
pos_(...)
 x. pos_() <==> +x
pow_(...)
 x. pow_(y[, z]) <==> pow(x, y[, z])
radd_(...)
 x. radd_(y) <==> y+x
rand_(...)
 x. rand_(y) <==> y&x
rdiv_(...)
 x. rdiv_(y) <==> y/x
rdivmod_(...)
 x. rdivmod_(y) <==> divmod(y, x)
reduce_(...)
rfloordiv_(...)
 x. rfloordiv_(y) <==> y//x
```

```
__rlshift__(...)
x.__rlshift__(y) <==> y<<x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__setstate__(...)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

all(...)
Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first

letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----  
`new_dtype : dtype`  
    New `'dtype'` object with the given change to the `byte` order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

sum(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

swapaxes(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

toString(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**var(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_\_array\_interface\_\_**  
Array protocol: Python side

**\_\_array\_priority\_\_**  
Array priority.

**\_\_array\_struct\_\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**dtype**  
get array data-descriptor

**flags**  
integer value of flags

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**

number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

**class unicode(builtin\_unicode, character)**

Method resolution order:

unicode  
builtin\_unicode  
builtin\_basestring  
character  
flexible  
generic  
builtin\_object

---

Methods defined here:

\_\_eq\_\_(...)  
x. \_\_eq\_\_(y) <==> x==y

\_\_ge\_\_(...)  
x. \_\_ge\_\_(y) <==> x>=y

\_\_gt\_\_(...)  
x. \_\_gt\_\_(y) <==> x>y

\_\_hash\_\_(...)  
x. \_\_hash\_\_() <==> hash(x)

\_\_le\_\_(...)  
x. \_\_le\_\_(y) <==> x<=y

\_\_lt\_\_(...)  
x. \_\_lt\_\_(y) <==> x<y

\_\_ne\_\_(...)  
x. \_\_ne\_\_(y) <==> x!=y

\_\_repr\_\_(...)  
x. \_\_repr\_\_() <==> repr(x)

\_\_str\_\_(...)  
x. \_\_str\_\_() <==> str(x)

---

Data and other attributes defined here:

\_\_new\_\_ = <built-in method \_\_new\_\_ of type object>  
T. \_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from builtin\_unicode:

\_\_add\_\_(...)  
x. \_\_add\_\_(y) <==> x+y

\_\_contains\_\_(...)  
x. \_\_contains\_\_(y) <==> y in x

\_\_format\_\_(...)  
S. \_\_format\_\_(format\_spec) -> unicode  
Return a formatted version of S as described by format\_spec.

\_\_getattribute\_\_(...)  
x. \_\_getattribute\_\_('name') <==> x.name

\_\_getitem\_\_(...)  
x. \_\_getitem\_\_(y) <==> x[y]

```
__getnewargs__(...)
__getslice__(...)
 x.__getslice__(i, j) <==> x[i:j]
 Use of negative indices is not supported.

__len__(...)
 x.__len__() <==> len(x)

__mod__(...)
 x.__mod__(y) <==> x%y

__mul__(...)
 x.__mul__(n) <==> x*n

__rmod__(...)
 x.__rmod__(y) <==> y%x

__rmul__(...)
 x.__rmul__(n) <==> n*x

__sizeof__(...)
 S.__sizeof__() -> size of S in memory, in bytes

capitalize(...)
 S.capitalize() -> unicode
 Return a capitalized version of S, i.e. make the first character
 have upper case and the rest lower case.

center(...)
 S.center(width[, fillchar]) -> unicode
 Return S centered in a Unicode string of length width. Padding is
 done using the specified fill character (default is a space)

count(...)
 S.count(sub[, start[, end]]) -> int
 Return the number of non-overlapping occurrences of substring sub in
 Unicode string S[start:end]. Optional arguments start and end are
 interpreted as in slice notation.

decode(...)
 S.decode([encoding[,errors]]) -> string or unicode
 Decodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'
 as well as any other name registered with codecs.register_error that is
 able to handle UnicodeDecodeErrors.

encode(...)
 S.encode([encoding[,errors]]) -> string or unicode
 Encodes S using the codec registered for encoding. encoding defaults
 to the default encoding. errors may be given to set a different error
 handling scheme. Default is 'strict' meaning that encoding errors raise
 a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
 'xmlcharrefreplace' as well as any other name registered with
 codecs.register_error that can handle UnicodeEncodeErrors.

endswith(...)
 S.endswith(suffix[, start[, end]]) -> bool
 Return True if S ends with the specified suffix, False otherwise.
 With optional start, test S beginning at that position.
 With optional end, stop comparing S at that position.
 suffix can also be a tuple of strings to try.

expandtabs(...)
 S.expandtabs([tabsize]) -> unicode
 Return a copy of S where all tab characters are expanded using spaces.
 If tabsize is not given, a tab size of 8 characters is assumed.

find(...)
 S.find(sub [,start [,end]]) -> int
```

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

**format(...)**  
S.format(\*args, \*\*kwargs) -> unicode

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

**index(...)**  
S.index(sub [,start [,end]]) -> int

Like S.find() but raise ValueError when the substring is not found.

**isalnum(...)**  
S.isalnum() -> bool

Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

**isalpha(...)**  
S.isalpha() -> bool

Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

**isdecimal(...)**  
S.isdecimal() -> bool

Return True if there are only decimal characters in S, False otherwise.

**isdigit(...)**  
S.isdigit() -> bool

Return True if all characters in S are digits and there is at least one character in S, False otherwise.

**islower(...)**  
S.islower() -> bool

Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

**isnumeric(...)**  
S.isnumeric() -> bool

Return True if there are only numeric characters in S, False otherwise.

**isspace(...)**  
S.isspace() -> bool

Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

**istitle(...)**  
S.istitle() -> bool

Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

**isupper(...)**  
S.isupper() -> bool

Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

**join(...)**  
S.join(iterable) -> unicode

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

**ljust(...)**

```
S.ljust(width[, fillchar]) -> int
Return S left-justified in a Unicode string of length width. Padding is
done using the specified fill character (default is a space).

lower(...)
S.lower() -> unicode

Return a copy of the string S converted to lowercase.

lstrip(...)
S.lstrip([chars]) -> unicode

Return a copy of the string S with leading whitespace removed.
If chars is given and not None, remove characters in chars instead.
If chars is a str, it will be converted to unicode before stripping

partition(...)
S.partition(sep) -> (head, sep, tail)

Search for the separator sep in S, and return the part before it,
the separator itself, and the part after it. If the separator is not
found, return S and two empty strings.

replace(...)
S.replace(old, new[, count]) -> unicode

Return a copy of S with all occurrences of substring
old replaced by new. If the optional argument count is
given, only the first count occurrences are replaced.

rfind(...)
S.rfind(sub [,start [,end]]) -> int

Return the highest index in S where substring sub is found,
such that sub is contained within S[start:end]. Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(...)
S.rindex(sub [,start [,end]]) -> int

Like S.rfind() but raise ValueError when the substring is not found.

rjust(...)
S.rjust(width[, fillchar]) -> unicode

Return S right-justified in a Unicode string of length width. Padding is
done using the specified fill character (default is a space).

rpartition(...)
S.rpartition(sep) -> (head, sep, tail)

Search for the separator sep in S, starting at the end of S, and return
the part before it, the separator itself, and the part after it. If the
separator is not found, return two empty strings and S.

rsplit(...)
S.rsplit([sep [,maxsplit]]) -> list of strings

Return a list of the words in S, using sep as the
delimiter string, starting at the end of the string and
working to the front. If maxsplit is given, at most maxsplit
splits are done. If sep is not specified, any whitespace string
is a separator.

rstrip(...)
S.rstrip([chars]) -> unicode

Return a copy of the string S with trailing whitespace removed.
If chars is given and not None, remove characters in chars instead.
If chars is a str, it will be converted to unicode before stripping

split(...)
S.split([sep [,maxsplit]]) -> list of strings

Return a list of the words in S, using sep as the
delimiter string. If maxsplit is given, at most maxsplit
splits are done. If sep is not specified or is None, any
whitespace string is a separator and empty strings are
removed from the result.
```

**splitlines(...)**  
`S.splitlines(keepends=False) -> list of strings`  
 Return a list of the lines in S, breaking at line boundaries.  
 Line breaks are not included in the resulting list unless keepends  
 is given and true.

**startswith(...)**  
`S.startswith(prefix[, start[, end]]) -> bool`  
 Return True if S starts with the specified prefix, False otherwise.  
 With optional start, test S beginning at that position.  
 With optional end, stop comparing S at that position.  
 prefix can also be a tuple of strings to try.

**strip(...)**  
`S.strip([chars]) -> unicode`  
 Return a copy of the string S with leading and trailing  
 whitespace removed.  
 If chars is given and not None, remove characters in chars instead.  
 If chars is a str, it will be converted to unicode before stripping

**swapcase(...)**  
`S.swapcase() -> unicode`  
 Return a copy of S with uppercase characters converted to lowercase  
 and vice versa.

**title(...)**  
`S.title() -> unicode`  
 Return a titlecased version of S, i.e. words start with title case  
 characters, all remaining cased characters have lower case.

**translate(...)**  
`S.translate(table) -> unicode`  
 Return a copy of the string S, where all characters have been mapped  
 through the given translation table, which must be a mapping of  
 Unicode ordinals to Unicode ordinals, Unicode strings or None.  
 Unmapped characters are left untouched. Characters mapped to None  
 are deleted.

**upper(...)**  
`S.upper() -> unicode`  
 Return a copy of S converted to uppercase.

**zfill(...)**  
`S.zfill(width) -> unicode`  
 Pad a numeric string S with zeros on the left, to fill a field  
 of the specified width. The string S is never truncated.

Methods inherited from [generic](#):

**\_\_abs\_\_(...)**  
`x.__abs__() <==> abs(x)`

**\_\_and\_\_(...)**  
`x.__and__(y) <==> x&y`

**\_\_array\_\_(...)**  
`sc.__array__(|type) return 0-dim array`

**\_\_array\_wrap\_\_(...)**  
`sc.__array_wrap__(obj) return scalar from array`

**\_\_copy\_\_(...)**

**\_\_deepcopy\_\_(...)**

**\_\_div\_\_(...)**  
`x.__div__(y) <==> x/y`

**\_\_divmod\_\_(...)**  
`x.__divmod__(y) <==> divmod(x, y)`

```
float(...)
x._float_() <==> float(x)

floordiv(...)
x._floordiv_(y) <==> x//y

hex(...)
x._hex_() <==> hex(x)

int(...)
x._int_() <==> int(x)

invert(...)
x._invert_() <==> ~x

long(...)
x._long_() <==> long(x)

lshift(...)
x._lshift_(y) <==> x<<y

neg(...)
x._neg_() <==> -x

nonzero(...)
x._nonzero_() <==> x != 0

oct(...)
x._oct_() <==> oct(x)

or(...)
x._or_(y) <==> x|y

pos(...)
x._pos_() <==> +x

pow(...)
x._pow_(y[, z]) <==> pow(x, y[, z])

radd(...)
x._radd_(y) <==> y+x

rand(...)
x._rand_(y) <==> y&x

rdiv(...)
x._rdiv_(y) <==> y/x

rdivmod(...)
x._rdivmod_(y) <==> divmod(y, x)

reduce(...)

rfloordiv(...)
x._rfloordiv_(y) <==> y//x

rlshift(...)
x._rlshift_(y) <==> y<<x

ror(...)
x._ror_(y) <==> y|x

rpow(...)
y._rpow_(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x._rrshift_(y) <==> y>>x

rshift(...)
x._rshift_(y) <==> x>>y

rsub(...)
x._rsub_(y) <==> y-x
```

```
__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argsort(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

astype(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**getfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of 'new\_order' for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`  
New [dtype](#) object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**prod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

toString(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

**\_\_array\_interface\_\_**

Array protocol: Python side

**\_\_array\_priority\_\_**

Array priority.

**\_\_array\_struct\_\_**

Array protocol: struct

**base**

base object

**data**

pointer to start of data

**dtype**

get array data-descriptor

**flags**

integer value of flags

**flat**

a 1-d view of scalar

**imag**

imaginary part of scalar

**itemsize**

length of one element in bytes

**nbytes**

length of item in bytes

**ndim**

number of array dimensions

**real**

real part of scalar

**shape**

tuple of array dimensions

**size**

number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

---

class [unsignedinteger](#)([integer](#))

Method resolution order:

[unsignedinteger](#)

[integer](#)

[number](#)

[generic](#)

[builtin\\_.object](#)

---

Data descriptors inherited from [integer](#):

**denominator**

denominator of value (1)

**numerator**

numerator of value (the value itself)

Methods inherited from [generic](#):**\_abs\_(...)**  
x. [\\_abs\\_](#)() <==> abs(x)**\_add\_(...)**  
x. [\\_add\\_](#)(y) <==> x+y**\_and\_(...)**  
x. [\\_and\\_](#)(y) <==> x&y**\_array\_(...)**  
sc. [\\_array\\_](#)(|type) return 0-dim array**\_array\_wrap\_(...)**  
sc. [\\_array\\_wrap\\_](#)(obj) return scalar from array**\_copy\_(...)****\_deepcopy\_(...)****\_div\_(...)**  
x. [\\_div\\_](#)(y) <==> x/y**\_divmod\_(...)**  
x. [\\_divmod\\_](#)(y) <==> divmod(x, y)**\_eq\_(...)**  
x. [\\_eq\\_](#)(y) <==> x==y**\_float\_(...)**  
x. [\\_float\\_](#)() <==> float(x)**\_floordiv\_(...)**  
x. [\\_floordiv\\_](#)(y) <==> x//y**\_format\_(...)**  
NumPy array scalar formatter**\_ge\_(...)**  
x. [\\_ge\\_](#)(y) <==> x>=y**\_getitem\_(...)**  
x. [\\_getitem\\_](#)(y) <==> x[y]**\_gt\_(...)**  
x. [\\_gt\\_](#)(y) <==> x>y**\_hex\_(...)**  
x. [\\_hex\\_](#)() <==> hex(x)**\_int\_(...)**  
x. [\\_int\\_](#)() <==> int(x)**\_invert\_(...)**  
x. [\\_invert\\_](#)() <==> ~x**\_le\_(...)**  
x. [\\_le\\_](#)(y) <==> x<=y**\_long\_(...)**  
x. [\\_long\\_](#)() <==> long(x)**\_lshift\_(...)**  
x. [\\_lshift\\_](#)(y) <==> x<<y**\_lt\_(...)**

```
x. lt_(y) <==> x<y
mod_(...)
x. mod_(y) <==> x%y
mul_(...)
x. mul_(y) <==> x*y
ne_(...)
x. ne_(y) <==> x!=y
neg_(...)
x. neg_()
<==> -x
nonzero_(...)
x. nonzero_()
<==> x != 0
oct_(...)
x. oct_()
<==> oct(x)
or_(...)
x. or_(y) <==> x|y
pos_(...)
x. pos_()
<==> +x
pow_(...)
x. pow_(y[, z]) <==> pow(x, y[, z])
radd_(...)
x. radd_(y) <==> y+x
rand_(...)
x. rand_(y) <==> y&x
rdiv_(...)
x. rdiv_(y) <==> y/x
rdivmod_(...)
x. rdivmod_(y) <==> divmod(y, x)
reduce_(...)
repr_(...)
x. repr_()
<==> repr(x)
rfloordiv_(...)
x. rfloordiv_(y) <==> y//x
rlshift_(...)
x. rlshift_(y) <==> y<<x
rmod_(...)
x. rmod_(y) <==> y%x
rmul_(...)
x. rmul_(y) <==> y*x
ror_(...)
x. ror_(y) <==> y|x
rpow_(...)
y. rpow_(x[, z]) <==> pow(x, y[, z])
rrshift_(...)
x. rrshift_(y) <==> y>>x
rshift_(...)
x. rshift_(y) <==> x>>y
rsub_(...)
x. rsub_(y) <==> y-x
rtruediv_(...)
```

```
x. rtruediv (y) <==> y/x
rxor(...)
x. rxor (y) <==> y^x
setstate(...)
sizeof(...)
str(...)
x. str () <==> str(x)
sub(...)
x. sub (y) <==> x-y
truediv(...)
x. truediv (y) <==> x/y
xor(...)
x. xor (y) <==> x^y

all(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

any(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmax(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argmin(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

argsort(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

astype(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**getfield(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New `dtype` object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

```
See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.
```

---

Data descriptors inherited from [generic](#):

**T**  
    transpose

**\_\_array\_interface\_\_**  
    Array protocol: Python side

**\_\_array\_priority\_\_**  
    Array priority.

**\_\_array\_struct\_\_**  
    Array protocol: struct

**base**  
    base object

**data**  
    pointer to start of data

**dtype**  
    get array data-descriptor

**flags**  
    integer value of flags

**flat**  
    a 1-d view of scalar

**imag**  
    imaginary part of scalar

**itemsize**  
    length of one element in bytes

**nbytes**  
    length of item in bytes

**ndim**  
    number of array dimensions

**real**  
    real part of scalar

**shape**  
    tuple of array dimensions

**size**  
    number of elements in the gentype

**strides**  
    tuple of bytes steps in each dimension

**ushort = class uint16([unsignedinteger](#))**  
Method resolution order:  
    [uint16](#)  
    [unsignedinteger](#)

---

[integer](#)  
[number](#)  
[generic](#)  
[builtin](#) [object](#)

---

Methods defined here:

---

[\\_\\_eq\\_\\_](#)(...)  
x. [\\_\\_eq\\_\\_](#)(y) <=> x==y

[\\_\\_ge\\_\\_](#)(...)  
x. [\\_\\_ge\\_\\_](#)(y) <=> x>=y

[\\_\\_gt\\_\\_](#)(...)  
x. [\\_\\_gt\\_\\_](#)(y) <=> x>y

[\\_\\_hash\\_\\_](#)(...)  
x. [\\_\\_hash\\_\\_](#)() <=> hash(x)

[\\_\\_index\\_\\_](#)(...)  
x[y:z] <=> x[y. [\\_\\_index\\_\\_](#)():z. [\\_\\_index\\_\\_](#)()]

[\\_\\_le\\_\\_](#)(...)  
x. [\\_\\_le\\_\\_](#)(y) <=> x<=y

[\\_\\_lt\\_\\_](#)(...)  
x. [\\_\\_lt\\_\\_](#)(y) <=> x<y

[\\_\\_ne\\_\\_](#)(...)  
x. [\\_\\_ne\\_\\_](#)(y) <=> x!=y

---

Data and other attributes defined here:

---

[\\_\\_new\\_\\_](#) = <built-in method [\\_\\_new\\_\\_](#) of type object>  
T. [\\_\\_new\\_\\_](#)(S, ...) -> a new [object](#) with type S, a subtype of T

---

Data descriptors inherited from [integer](#):

**denominator**  
denominator of value (1)

**numerator**  
numerator of value (the value itself)

---

Methods inherited from [generic](#):

---

[\\_\\_abs\\_\\_](#)(...)  
x. [\\_\\_abs\\_\\_](#)() <=> abs(x)

[\\_\\_add\\_\\_](#)(...)  
x. [\\_\\_add\\_\\_](#)(y) <=> x+y

[\\_\\_and\\_\\_](#)(...)  
x. [\\_\\_and\\_\\_](#)(y) <=> x&y

[\\_\\_array\\_\\_](#)(...)  
sc. [\\_\\_array\\_\\_](#)(|type) return 0-dim array

[\\_\\_array\\_wrap\\_\\_](#)(...)  
sc. [\\_\\_array\\_wrap\\_\\_](#)(obj) return scalar from array

[\\_\\_copy\\_\\_](#)(...)

[\\_\\_deepcopy\\_\\_](#)(...)

[\\_\\_div\\_\\_](#)(...)  
x. [\\_\\_div\\_\\_](#)(y) <=> x/y

[\\_\\_divmod\\_\\_](#)(...)  
x. [\\_\\_divmod\\_\\_](#)(y) <=> divmod(x, y)

```
float(...)
x.float() <==> float(x)

floordiv(...)
x.floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

getitem(...)
x.getitem(y) <==> x[y]

hex(...)
x.hex() <==> hex(x)

int(...)
x.int() <==> int(x)

invert(...)
x.invert() <==> ~x

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

neg(...)
x.neg() <==> -x

nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x
```

```
__rmod__(...)
 x. __rmod__(y) <==> y%x

__rmul__(...)
 x. __rmul__(y) <==> y*x

__ror__(...)
 x. __ror__(y) <==> y|x

__rpow__(...)
 y. __rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
 x. __rrshift__(y) <==> y>>x

__rshift__(...)
 x. __rshift__(y) <==> x>>y

__rsub__(...)
 x. __rsub__(y) <==> y-x

__rtruediv__(...)
 x. __rtruediv__(y) <==> y/x

__rxor__(...)
 x. __rxor__(y) <==> y^x

__setstate__(...)

__sizeof__(...)

__str__(...)
 x. __str__() <==> str(x)

__sub__(...)
 x. __sub__(y) <==> x-y

__truediv__(...)
 x. __truediv__(y) <==> x/y

__xor__(...)
 x. __xor__(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumprod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**cumsum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**diagonal(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dump(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**dumps(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**fill(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**flatten(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses,

```
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

getfield(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

item(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

itemset(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

max(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

mean(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

min(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

newbyteorder(...)
newbyteorder(new_order='S')

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
```

```
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

Parameters

new_order : str, optional
 Byte order to force; a value from the byte order specifications
 above. The default value ('S') results in swapping the current
 byte order. The code does a case-insensitive check on the first
 letter of `new_order` for the alternatives above. For example,
 any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

new_dtype : dtype
 New dtype object with the given change to the byte order.

nonzero(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

prod(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ptp(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

put(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

ravel(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

repeat(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

reshape(...)
```

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**resize(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setfield(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tostring(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

### **view(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

#### **T**

transpose

#### **\_\_array\_interface\_\_**

Array protocol: Python side

#### **\_\_array\_priority\_\_**

Array priority.

#### **\_\_array\_struct\_\_**

Array protocol: struct

#### **base**

base object

#### **data**

pointer to start of data

#### **dtype**

get array data-descriptor

#### **flags**

integer value of flags

#### **flat**

a 1-d view of scalar

#### **imag**

imaginary part of scalar

#### **itemsize**

length of one element in bytes

#### **nbytes**

length of item in bytes

```

ndim
 number of array dimensions

real
 real part of scalar

shape
 tuple of array dimensions

size
 number of elements in the gentype

strides
 tuple of bytes steps in each dimension

class vectorize(builtin_object)
 vectorize(pyfunc, otypes='', doc=None, excluded=None, cache=False)

 Generalized function class.

 Define a vectorized function which takes a nested sequence
 of objects or numpy arrays as inputs and returns a
 numpy array as output. The vectorized function evaluates `pyfunc` over
 successive tuples of the input arrays like the python map function,
 except it uses the broadcasting rules of numpy.

 The data type of the output of `vectorized` is determined by calling
 the function with the first element of the input. This can be avoided
 by specifying the `otypes` argument.

 Parameters

 pyfunc : callable
 A python function or method.
 otypes : str or list of dtypes, optional
 The output data type. It must be specified as either a string of
 typecode characters or a list of data type specifiers. There should
 be one data type specifier for each output.
 doc : str, optional
 The docstring for the function. If `None`, the docstring will be the
 ``pyfunc.__doc__``.
 excluded : set, optional
 Set of strings or integers representing the positional or keyword
 arguments for which the function will not be vectorized. These will be
 passed directly to `pyfunc` unmodified.

 .. versionadded:: 1.7.0

 cache : bool, optional
 If `True`, then cache the first function call that determines the number
 of outputs if `otypes` is not provided.

 .. versionadded:: 1.7.0

 Returns

 vectorized : callable
 Vectorized function.

 Examples

 >>> def myfunc(a, b):
 ... "Return a-b if a>b, otherwise return a+b"
 ... if a > b:
 ... return a - b
 ... else:
 ... return a + b

 >>> vfunc = np.vectorize(myfunc)
 >>> vfunc([1, 2, 3, 4], 2)
 array([3, 4, 1, 2])

 The docstring is taken from the input function to `vectorize` unless it
 is specified

 >>> vfunc.__doc__
 'Return a-b if a>b, otherwise return a+b'
 >>> vfunc = np.vectorize(myfunc, doc='Vectorized `myfunc`')
 >>> vfunc.__doc__
 'Vectorized `myfunc`'

 The output type is determined by evaluating the first element of the input,
 unless it is specified

```

```
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<type 'numpy.int32'>
>>> vfunc = np.vectorize(myfunc, otypes=[np.float])
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<type 'numpy.float64'>
```

The `excluded` argument can be used to prevent vectorizing over certain arguments. This can be useful for array-like arguments of a fixed length such as the coefficients for a polynomial as in `polyval`:

```
>>> def mypolyval(p, x):
... _p = list(p)
... res = _p.pop(0)
... while _p:
... res = res*x + _p.pop(0)
... return res
>>> vpolyval = np.vectorize(mypolyval, excluded=['p'])
>>> vpolyval(p=[1, 2, 3], x=[0, 1])
array([3, 6])
```

Positional arguments may also be excluded by specifying their position:

```
>>> vpolyval.excluded.add(0)
>>> vpolyval([1, 2, 3], x=[0, 1])
array([3, 6])
```

#### Notes

The `vectorize` function is provided primarily for convenience, not for performance. The implementation is essentially a for loop.

If `otypes` is not specified, then a call to the function with the first argument will be used to determine the [number](#) of outputs. The results of this call will be cached if `cache` is `True` to prevent calling the function twice. However, to implement the cache, the original function must be wrapped which will slow down subsequent calls, so only do this if your function is expensive.

The new keyword argument interface and `excluded` argument support further degrades performance.

Methods defined here:

---

<a href="#"><u>__call__</u></a> (self, *args, **kwargs)	Return arrays with the results of `pyfunc` <a href="#">broadcast</a> (vectorized) over `args` and `kwargs` not in `excluded`.
<a href="#"><u>__init__</u></a> (self, pyfunc, otypes='', doc=None, excluded=None, cache=False)	

---

Data descriptors defined here:

---

<a href="#"><u>__dict__</u></a>	dictionary for instance variables (if defined)
<a href="#"><u>__weakref__</u></a>	list of weak references to the object (if defined)

---

class **void(flexible)**

Method resolution order:

<a href="#"><u>void</u></a>
<a href="#"><u>flexible</u></a>
<a href="#"><u>generic</u></a>
<a href="#"><u>builtin</u></a>
<a href="#"><u>_object</u></a>

---

Methods defined here:

---

<a href="#"><u>__delitem__</u></a> (...)	x. <a href="#"><u>__delitem__</u></a> (y) <==> del x[y]
<a href="#"><u>__eq__</u></a> (...)	x. <a href="#"><u>__eq__</u></a> (y) <==> x==y
<a href="#"><u>__ge__</u></a> (...)	x. <a href="#"><u>__ge__</u></a> (y) <==> x>=y

---

---

**\_\_getitem\_\_**(...)  
 x.\_\_getitem\_\_(y) <==> x[y]  
  
**\_\_gt\_\_**(...)  
 x.\_\_gt\_\_(y) <==> x>y  
  
**\_\_hash\_\_**(...)  
 x.\_\_hash\_\_() <==> hash(x)  
  
**\_\_le\_\_**(...)  
 x.\_\_le\_\_(y) <==> x<=y  
  
**\_\_len\_\_**(...)  
 x.\_\_len\_\_() <==> len(x)  
  
**\_\_lt\_\_**(...)  
 x.\_\_lt\_\_(y) <==> x<y  
  
**\_\_ne\_\_**(...)  
 x.\_\_ne\_\_(y) <==> x!=y  
  
**\_\_setitem\_\_**(...)  
 x.\_\_setitem\_\_(i, y) <==> x[i]=y  
  
**getfield**(...)  
  
**setfield**(...)

---

Data descriptors defined here:

**dtype**  
 dtype object

**flags**  
 integer value of flags

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
 T.\_\_new\_\_(S, ...) -> a new object with type S, a subtype of T

---

Methods inherited from generic:

**\_\_abs\_\_**(...)  
 x.\_\_abs\_\_() <==> abs(x)  
  
**\_\_add\_\_**(...)  
 x.\_\_add\_\_(y) <==> x+y  
  
**\_\_and\_\_**(...)  
 x.\_\_and\_\_(y) <==> x&y  
  
**\_\_array\_\_**(...)  
 sc.\_\_array\_\_(|type) return 0-dim array  
  
**\_\_array\_wrap\_\_**(...)  
 sc.\_\_array\_wrap\_\_(obj) return scalar from array  
  
**\_\_copy\_\_**(...)  
  
**\_\_deepcopy\_\_**(...)  
  
**\_\_div\_\_**(...)  
 x.\_\_div\_\_(y) <==> x/y  
  
**\_\_divmod\_\_**(...)  
 x.\_\_divmod\_\_(y) <==> divmod(x, y)  
  
**\_\_float\_\_**(...)  
 x.\_\_float\_\_() <==> float(x)  
  
**\_\_floordiv\_\_**(...)

```
x. floordiv (y) <==> x//y
format (...)
 NumPy array scalar formatter
hex (...)
 x. hex () <==> hex(x)
int (...)
 x. int () <==> int(x)
invert (...)
 x. invert () <==> ~x
long (...)
 x. long () <==> long(x)
lshift (...)
 x. lshift (y) <==> x<<y
mod (...)
 x. mod (y) <==> x%y
mul (...)
 x. mul (y) <==> x*y
neg (...)
 x. neg () <==> -x
nonzero (...)
 x. nonzero () <==> x != 0
oct (...)
 x. oct () <==> oct(x)
or (...)
 x. or (y) <==> x|y
pos (...)
 x. pos () <==> +x
pow (...)
 x. pow (y[, z]) <==> pow(x, y[, z])
radd (...)
 x. radd (y) <==> y+x
rand (...)
 x. rand (y) <==> y&x
rdiv (...)
 x. rdiv (y) <==> y/x
rdivmod (...)
 x. rdivmod (y) <==> divmod(y, x)
reduce (...)
repr (...)
 x. repr () <==> repr(x)
rfloordiv (...)
 x. rfloordiv (y) <==> y//x
rlshift (...)
 x. rlshift (y) <==> y<<x
rmod (...)
 x. rmod (y) <==> y%x
rmul (...)
 x. rmul (y) <==> y*x
ror (...)
```

```
x. __ror__(y) <==> y|x
__rpow__(...) y. rpow(x[, z]) <==> pow(x, y[, z])
__rrshift__(...) x. rrshift(y) <==> y>>x
__rshift__(...) x. rshift(y) <==> x>>y
__rsub__(...) x. rsub(y) <==> y-x
__rtruediv__(...) x. rtruediv(y) <==> y/x
__rxor__(...) x. rxor(y) <==> y^x
__setstate__(...)
__sizeof__(...)
__str__(...) x. str() <==> str(x)
__sub__(...) x. sub(y) <==> x-y
__truediv__(...) x. truediv(y) <==> x/y
__xor__(...) x. xor(y) <==> x^y
all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.
```

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**itemset(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**max(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**mean(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**min(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**newbyteorder(...)**

[newbyteorder](#)(new\_order='S')

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters

-----

`new_order : str, optional`

Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

-----

`new_dtype : dtype`

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**round(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**searchsorted(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**setflags(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**squeeze(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**std(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**sum(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**swapaxes(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

```

The corresponding attribute of the derived class of interest.

take(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tobytes(...)

tofile(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

tolist(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

toString(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

trace(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

transpose(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

var(...)
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses,
albeit unimplemented, all the attributes of the ndarray class
so as to provide a uniform API.

See Also

The corresponding attribute of the derived class of interest.

view(...)
Not implemented (virtual attribute)
```

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**

transpose

**\_\_array\_interface\_\_**

Array protocol: Python side

**\_\_array\_priority\_\_**

Array priority.

**\_\_array\_struct\_\_**

Array protocol: struct

**base**

base object

**data**

pointer to start of data

**flat**

a 1-d view of scalar

**imag**

imaginary part of scalar

**itemsize**

length of one element in bytes

**nbytes**

length of item in bytes

**ndim**

number of array dimensions

**real**

real part of scalar

**shape**

tuple of array dimensions

**size**

number of elements in the gentype

**strides**

tuple of bytes steps in each dimension

---

**void0** = class void([flexible](#))

Method resolution order:

[void](#)  
[flexible](#)  
[generic](#)  
[builtin](#) .[object](#)

---

Methods defined here:

**\_\_delitem\_\_(...)**

x. [\\_\\_delitem\\_\\_](#)(y) <==> del x[y]

**\_\_eq\_\_(...)**

x. [\\_\\_eq\\_\\_](#)(y) <==> x==y

---

**\_\_ge\_\_(...)**  
 $x.\underline{\text{ge}}(y) \iff x \geq y$   
**\_\_getitem\_\_(...)**  
 $x.\underline{\text{getitem}}(y) \iff x[y]$   
**\_\_gt\_\_(...)**  
 $x.\underline{\text{gt}}(y) \iff x > y$   
**\_\_hash\_\_(...)**  
 $x.\underline{\text{hash}}() \iff \text{hash}(x)$   
**\_\_le\_\_(...)**  
 $x.\underline{\text{le}}(y) \iff x \leq y$   
**\_\_len\_\_(...)**  
 $x.\underline{\text{len}}() \iff \text{len}(x)$   
**\_\_lt\_\_(...)**  
 $x.\underline{\text{lt}}(y) \iff x < y$   
**\_\_ne\_\_(...)**  
 $x.\underline{\text{ne}}(y) \iff x \neq y$   
**\_\_setitem\_\_(...)**  
 $x.\underline{\text{setitem}}(i, y) \iff x[i] = y$   
**getfield(...)**  
**setfield(...)**

---

Data descriptors defined here:

**dtype**  
 $\text{dtype object}$

**flags**  
 $\text{integer value of flags}$

---

Data and other attributes defined here:

**\_\_new\_\_** = <built-in method \_\_new\_\_ of type object>  
 $T.\underline{\text{new}}(S, \dots) \rightarrow \text{a new } \underline{\text{object}} \text{ with type } S, \text{ a subtype of } T$

---

Methods inherited from generic:

**\_\_abs\_\_(...)**  
 $x.\underline{\text{abs}}() \iff \text{abs}(x)$   
**\_\_add\_\_(...)**  
 $x.\underline{\text{add}}(y) \iff x + y$   
**\_\_and\_\_(...)**  
 $x.\underline{\text{and}}(y) \iff x \& y$   
**\_\_array\_\_(...)**  
 $\text{sc.}\underline{\text{array}}(|\text{type}|) \text{ return 0-dim array}$   
**\_\_array\_wrap\_\_(...)**  
 $\text{sc.}\underline{\text{array wrap}}(\text{obj}) \text{ return scalar from array}$   
**\_\_copy\_\_(...)**  
**\_\_deepcopy\_\_(...)**  
**\_\_div\_\_(...)**  
 $x.\underline{\text{div}}(y) \iff x / y$   
**\_\_divmod\_\_(...)**  
 $x.\underline{\text{divmod}}(y) \iff \text{divmod}(x, y)$   
**\_\_float\_\_(...)**

```
x.float() <==> float(x)

floordiv(...)
x.floordiv(y) <==> x//y

format(...)
NumPy array scalar formatter

hex(...)
x.hex() <==> hex(x)

int(...)
x.int() <==> int(x)

invert(...)
x.invert() <==> ~x

long(...)
x.long() <==> long(x)

lshift(...)
x.lshift(y) <==> x<<y

mod(...)
x.mod(y) <==> x%y

mul(...)
x.mul(y) <==> x*y

neg(...)
x.neg() <==> -x

nonzero(...)
x.nonzero() <==> x != 0

oct(...)
x.oct() <==> oct(x)

or(...)
x.or(y) <==> x|y

pos(...)
x.pos() <==> +x

pow(...)
x.pow(y[, z]) <==> pow(x, y[, z])

radd(...)
x.radd(y) <==> y+x

rand(...)
x.rand(y) <==> y&x

rdiv(...)
x.rdiv(y) <==> y/x

rdivmod(...)
x.rdivmod(y) <==> divmod(y, x)

reduce(...)

repr(...)
x.repr() <==> repr(x)

rfloordiv(...)
x.rfloordiv(y) <==> y//x

rlshift(...)
x.rlshift(y) <==> y<<x

rmod(...)
x.rmod(y) <==> y%y

rmul(...)
```

```
x.rmul(y) <==> y*x
ror(...)
x.ror(y) <==> y|x

rpow(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

rrshift(...)
x.rrshift(y) <==> y>>x

rshift(...)
x.rshift(y) <==> x>>y

rsub(...)
x.rsub(y) <==> y-x

rtruediv(...)
x.rtruediv(y) <==> y/x

rxor(...)
x.rxor(y) <==> y^x

setstate(...)

sizeof(...)

str(...)
x.str() <==> str(x)

sub(...)
x.sub(y) <==> x-y

truediv(...)
x.truediv(y) <==> x/y

xor(...)
x.xor(y) <==> x^y

all(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

any(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmax(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.

 See Also

 The corresponding attribute of the derived class of interest.

argmin(...)
 Not implemented (virtual attribute)

 Class generic exists solely to derive numpy scalars from, and possesses,
 albeit unimplemented, all the attributes of the ndarray class
 so as to provide a uniform API.
```

See Also  
-----  
The corresponding attribute of the derived class of interest.

**argsort(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**astype(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**byteswap(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**choose(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**clip(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**compress(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**conj(...)**

**conjugate(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**copy(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumprod(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**cumsum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**diagonal(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dump(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**dumps(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**fill(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**flatten(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**item(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**itemset(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**max(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**mean(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**min(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**newbyteorder(...)**  
[newbyteorder\(new\\_order='S'\)](#)

Return a new `dtype` with a different [byte](#) order.

Changes are also made in all fields and sub-arrays of the data type.

The `new\_order` code can be any from the following:

\* 'S' - swap [dtype](#) from current to opposite endian  
\* {'<', 'L'} - little endian  
\* {'>', 'B'} - big endian  
\* {'=', 'N'} - native order  
\* {'|', 'I'} - ignore (no change to [byte](#) order)

Parameters  
-----  
**new\_order : str, optional**  
Byte order to force; a value from the [byte](#) order specifications above. The default value ('S') results in swapping the current [byte](#) order. The code does a case-insensitive check on the first letter of `new\_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns  
-----  
**new\_dtype : dtype**

New `dtype` object with the given change to the [byte](#) order.

**nonzero(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**prod(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ptp(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**put(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**ravel(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**repeat(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**reshape(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**resize(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class

so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**round(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**searchsorted(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**setflags(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sort(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**squeeze(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**std(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**sum(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**swapaxes(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**take(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tobytes(...)**

**tofile(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**tolist(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**toString(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**trace(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**transpose(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also

-----

The corresponding attribute of the derived class of interest.

**var(...)**

Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

**view(...)**  
Not implemented (virtual attribute)

Class [generic](#) exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the [ndarray](#) class so as to provide a uniform API.

See Also  
-----  
The corresponding attribute of the derived class of interest.

---

Data descriptors inherited from [generic](#):

**T**  
transpose

**\_array\_interface\_**  
Array protocol: Python side

**\_array\_priority\_**  
Array priority.

**\_array\_struct\_**  
Array protocol: struct

**base**  
base object

**data**  
pointer to start of data

**flat**  
a 1-d view of scalar

**imag**  
imaginary part of scalar

**itemsize**  
length of one element in bytes

**nbytes**  
length of item in bytes

**ndim**  
number of array dimensions

**real**  
real part of scalar

**shape**  
tuple of array dimensions

**size**  
number of elements in the gentype

**strides**  
tuple of bytes steps in each dimension

## Functions

**add\_docstring(...)**  
[add\\_docstring](#)(obj, docstring)

Add a docstring to a built-in obj if possible.  
If the obj already has a docstring raise a [RuntimeError](#)  
If this routine does not know how to add a docstring to the [object](#)  
raise a TypeError

**add\_newdoc(place, obj, doc)**  
Adds documentation to obj which is in module place.

```
If doc is a string add it to obj as a docstring
If doc is a tuple, then the first element is interpreted as
an attribute of obj and the second as the docstring
(method, docstring)

If doc is a list, then each element of the list should be a
sequence of length two --> [(method1, docstring1),
(method2, docstring2), ...]

This routine never raises an error.

This routine cannot modify read-only docstrings, as appear
in new-style classes or built-in functions. Because this
routine never raises an error the caller must check manually
that the docstrings were changed.

add_newdoc_ufunc = _add_newdoc_ufunc(...)
 add_ufunc_docstring(ufunc, new_docstring)

Replace the docstring for a ufunc with new_docstring.
This method will only work if the current docstring for
the ufunc is NULL. (At the C level, i.e. when ufunc->doc is NULL.)

Parameters

ufunc : numpy.ufunc
 A ufunc whose current doc is NULL.
new_docstring : string
 The new docstring for the ufunc.
```

Notes

-----

```
This method allocates memory for new_docstring on
the heap. Technically this creates a memory leak, since this
memory will not be reclaimed until the end of the program
even if the ufunc itself is removed. However this will only
be a problem if the user is repeatedly creating ufuncs with
no documentation, adding documentation via add_newdoc_ufunc,
and then throwing away the ufunc.
```

**alen(a)**

```
Return the length of the first dimension of the input array.
```

Parameters

-----

```
a : array_like
 Input array.
```

Returns

-----

```
alen : int
 Length of the first dimension of `a`.
```

See Also

-----

```
shape, size
```

Examples

-----

```
>>> a = np.zeros((7,4,5))
>>> a.shape[0]
7
>>> np.alen(a)
7
```

**all(a, axis=None, out=None, keepdims=False)**

```
Test whether all array elements along a given axis evaluate to True.
```

Parameters

-----

```
a : array_like
 Input array or object that can be converted to an array.
axis : None or int or tuple of ints, optional
 Axis or axes along which a logical AND reduction is performed.
 The default ('axis' = 'None') is to perform a logical AND over all
 the dimensions of the input array. 'axis' may be negative, in
 which case it counts from the last to the first axis.

.. versionadded:: 1.7.0

 If this is a tuple of ints, a reduction is performed on multiple
 axes, instead of a single axis or all the axes as before.
out : ndarray, optional
 Alternate output array in which to place the result.
 It must have the same shape as the expected output and its
 type is preserved (e.g., if ``dtype(out)`` is float, the result
```

```

will consist of 0.0's and 1.0's). See `doc.ufuncs` (Section
"Output arguments") for more details.

keepdims : bool, optional
 If this is set to True, the axes which are reduced are left
 in the result as dimensions with size one. With this option,
 the result will broadcast correctly against the original `arr`.

Returns

all : ndarray, bool
 A new boolean or array is returned unless `out` is specified,
 in which case a reference to `out` is returned.

See Also

ndarray.all : equivalent method

any : Test whether any element along a given axis evaluates to True.

Notes

Not a Number (NaN), positive infinity and negative infinity
evaluate to 'True' because these are not equal to zero.

Examples

>>> np.all([[True,False],[True,True]])
False

>>> np.all([[True,False],[True,True]], axis=0)
array([[True, False], dtype=bool])

>>> np.all([-1, 4, 5])
True

>>> np.all([1.0, np.nan])
True

>>> o=np.array([False])
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
doctest: +SKIP
(28293632, 28293632, array([True], dtype=bool))

allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
Returns True if two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The
relative difference (`rtol` * abs(`b`)) and the absolute difference
`atol` are added together to compare against the absolute difference
between `a` and `b`.

If either array contains one or more NaNs, False is returned.
Infs are treated as equal if they are in the same place and of the same
sign in both arrays.

Parameters

a, b : array_like
 Input arrays to compare.
rtol : float
 The relative tolerance parameter (see Notes).
atol : float
 The absolute tolerance parameter (see Notes).
equal_nan : bool
 Whether to compare NaN's as equal. If True, NaN's in `a` will be
 considered equal to NaN's in `b` in the output array.

.. versionadded:: 1.10.0

Returns

allclose : bool
 Returns True if the two arrays are equal within the given
 tolerance; False otherwise.

See Also

isclose, all, any

Notes

If the following equation is element-wise True, then allclose returns
True.

 absolute(`a` - `b`) <= (`atol` + `rtol` * absolute(`b`))

The above equation is not symmetric in `a` and `b`, so that
`allclose(a, b)` might be different from `allclose(b, a)` in
some rare cases.

```

Examples

```
----->>> np.allclose([1e10,1e-7], [1.00001e10,1e-8])
False
>>> np.allclose([1e10,1e-8], [1.00001e10,1e-9])
True
>>> np.allclose([1e10,1e-8], [1.0001e10,1e-9])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
True
```

**alltrue(a, axis=None, out=None, keepdims=False)**

Check if all elements of input array are true.

See Also

```
-----numpy.all : Equivalent function; see for details.
```

**alterdot()**

Change `dot`, `vdot`, and `inner` to use accelerated BLAS functions.

Typically, as a user of Numpy, you do not explicitly call this function. If Numpy is built with an accelerated BLAS, this function is automatically called when Numpy is imported.

When Numpy is built with an accelerated BLAS like ATLAS, these functions are replaced to make use of the faster implementations. The faster implementations only affect `float32`, `float64`, `complex64`, and `complex128` arrays. Furthermore, the BLAS API only includes `matrix-matrix`, `matrix-vector`, and vector-vector products. Products of arrays with larger dimensionalities use the built in functions and are not accelerated.

.. note:: Deprecated in Numpy 1.10  
The cblas functions have been integrated into the multarray module and alterdot now longer does anything. It will be removed in Numpy 1.11.0.

See Also

```
-----restoredot : `restoredot` undoes the effects of `alterdot`.
```

**amax(a, axis=None, out=None, keepdims=False)**

Return the maximum of an array or maximum along an axis.

Parameters

```
-----a : array_like
 Input data.
axis : None or int or tuple of ints, optional
 Axis or axes along which to operate. By default, flattened input is used.

.. versionadded: 1.7.0

If this is a tuple of ints, the maximum is selected over multiple axes,
instead of a single axis or all the axes as before.
out : ndarray, optional
 Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
 See 'doc.ufuncs' (Section "Output arguments") for more details.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original 'arr'.
```

Returns

```
-----amax : ndarray or scalar
 Maximum of 'a'. If 'axis' is None, the result is a scalar value.
 If 'axis' is given, the result is an array of dimension
 ``a.ndim - 1``.
```

See Also

```
-----amin :
 The minimum value of an array along a given axis, propagating any NaNs.
nanmax :
 The maximum value of an array along a given axis, ignoring any NaNs.
maximum :
 Element-wise maximum of two arrays, propagating any NaNs.
fmax :
 Element-wise maximum of two arrays, ignoring any NaNs.
argmax :
 Return the indices of the maximum values.
```

```

nanmin, minimum, fmin

Notes

NaN values are propagated, that is if at least one item is NaN, the
corresponding min value will be NaN as well. To ignore NaN values
(MATLAB behavior), please use nanmin.

Don't use `amin` for element-wise comparison of 2 arrays; when
``a.shape[0]`` is 2, ``minimum(a[0], a[1])`` is faster than
``amin(a, axis=0)``.

Examples

>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
 [2, 3]])
>>> np.amax(a) # Maximum of the flattened array
3
>>> np.amax(a, axis=0) # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1) # Maxima along the second axis
array([1, 3])

>>> b = np.arange(5, dtype=np.float)
>>> b[2] = np.NaN
>>> np.amax(b)
nan
>>> np.nanmax(b)
4.0

```

**amin(a, axis=None, out=None, keepdims=False)**

Return the minimum of an array or minimum along an axis.

Parameters

a : array\_like  
Input data.

axis : None or [int](#) or tuple of ints, optional  
Axis or axes along which to operate. By default, flattened input is used.

.. versionadded: 1.7.0

If this is a tuple of ints, the minimum is selected over multiple axes, instead of a [single](#) axis or all the axes as before.

out : [ndarray](#), optional  
Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See [`doc.ufuncs`](#) (Section "Output arguments") for more details.

keepdims : bool, optional  
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will [broadcast](#) correctly against the original `arr`.

Returns

amin : [ndarray](#) or scalar  
Minimum of `a`. If `axis` is None, the result is a scalar value. If `axis` is given, the result is an array of dimension ``a.ndim - 1``.

See Also

amax : The maximum value of an array along a given axis, propagating any NaNs.

nanmin : The minimum value of an array along a given axis, ignoring any NaNs.

minimum : Element-wise minimum of two arrays, propagating any NaNs.

fmin : Element-wise minimum of two arrays, ignoring any NaNs.

argmin : Return the indices of the minimum values.

nanmax, maximum, fmax

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don't use `amin` for element-wise comparison of 2 arrays; when ``a.shape[0]`` is 2, ``minimum(a[0], a[1])`` is faster than ``[amin](#)(a, axis=0)``.

**Examples**

```

>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
 [2, 3]])
>>> np.amin(a) # Minimum of the flattened array
0
>>> np.amin(a, axis=0) # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1) # Minima along the second axis
array([0, 2])

>>> b = np.arange(5, dtype=np.float)
>>> b[2] = np.NaN
>>> np.amin(b)
nan
>>> np.nanmin(b)
0.0
```

**angle(z, deg=0)**

Return the angle of the [complex](#) argument.

**Parameters**

```

z : array_like
 A complex number or sequence of complex numbers.
deg : bool, optional
 Return angle in degrees if True, radians if False (default).
```

**Returns**

```

angle : ndarray or scalar
 The counterclockwise angle from the positive real axis on
 the complex plane, with dtype as numpy.float64.
```

**See Also**

```

arctan2
absolute
```

**Examples**

```

>>> np.angle([1.0, 1.0j, 1+1j]) # in radians
array([0. , 1.57079633, 0.78539816])
>>> np.angle(1+1j, deg=True) # in degrees
45.0
```

**any(a, axis=None, out=None, keepdims=False)**

Test whether any array element along a given axis evaluates to True.

**Returns** [single](#) boolean unless `axis` is not ``None``

**Parameters**

```

a : array_like
 Input array or object that can be converted to an array.
axis : None or int or tuple of ints, optional
 Axis or axes along which a logical OR reduction is performed.
 The default ('axis' = 'None') is to perform a logical OR over all
 the dimensions of the input array. 'axis' may be negative, in
 which case it counts from the last to the first axis.

.. versionadded:: 1.7.0

 If this is a tuple of ints, a reduction is performed on multiple
 axes, instead of a single axis or all the axes as before.
out : ndarray, optional
 Alternate output array in which to place the result. It must have
 the same shape as the expected output and its type is preserved
 (e.g., if it is of type float, then it will remain so, returning
 1.0 for True and 0.0 for False, regardless of the type of 'a').
 See 'doc.ufuncs' (Section "Output arguments") for details.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left
 in the result as dimensions with size one. With this option,
 the result will broadcast correctly against the original 'arr'.

Returns

any : bool or ndarray
 A new boolean or 'ndarray' is returned unless 'out' is specified,
 in which case a reference to 'out' is returned.
```

**See Also**

```

```

[`ndarray.any`](#) : equivalent method

`all` : Test whether all elements along a given axis evaluate to True.

**Notes**  
 Not a Number (NaN), positive infinity and negative infinity evaluate to 'True' because these are not equal to zero.

**Examples**

```
>>> np.any([[True, False], [True, True]])
True

>>> np.any([[True, False], [False, False]], axis=0)
array([True, False], dtype=bool)

>>> np.any([-1, 0, 5])
True

>>> np.any(np.nan)
True

>>> o=np.array([False])
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array([True], dtype=bool), array([True], dtype=bool))
>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o) # identity of z and o # doctest: +SKIP
(191614240, 191614240)
```

**append(arr, values, axis=None)**

Append values to the end of an array.

**Parameters**

`arr` : array\_like  
 Values are appended to a copy of this array.  
`values` : array\_like  
 These values are appended to a copy of `arr`. It must be of the correct shape (the same shape as `arr`, excluding `axis`). If `axis` is not specified, `values` can be any shape and will be flattened before use.  
`axis` : int, optional  
 The axis along which `values` are appended. If `axis` is not given, both `arr` and `values` are flattened before use.

**Returns**

`append` : [`ndarray`](#)  
 A copy of `arr` with `values` appended to `axis`. Note that `append` does not occur in-place: a new array is allocated and filled. If `axis` is None, `out` is a flattened array.

**See Also**

`insert` : Insert elements into an array.  
`delete` : Delete elements from an array.

**Examples**

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

When 'axis' is specified, 'values' must have the correct shape.

>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
>>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)
Traceback (most recent call last):
...
ValueError: arrays must have same number of dimensions
```

**apply\_along\_axis(func1d, axis, arr, \*args, \*\*kwargs)**

Apply a function to 1-D slices along the given axis.

Execute `func1d(a, \*args)` where `func1d` operates on 1-D arrays and `a` is a 1-D slice of `arr` along `axis`.

**Parameters**

`func1d` : function  
 This function should accept 1-D arrays. It is applied to 1-D slices of `arr` along the specified axis.

```

axis : integer
 Axis along which `arr` is sliced.
arr : ndarray
 Input array.
args : any
 Additional arguments to `func1d`.
kwargs: any
 Additional named arguments to `func1d`.

.. versionadded:: 1.9.0

>Returns

apply_along_axis : ndarray
 The output array. The shape of `outarr` is identical to the shape of
 `arr`, except along the `axis` dimension, where the length of `outarr`
 is equal to the size of the return value of `func1d`. If `func1d`
 returns a scalar `outarr` will have one fewer dimensions than `arr`.

See Also

apply_over_axes : Apply a function repeatedly over multiple axes.

Examples

>>> def my_func(a):
... """Average first and last element of a 1-D array"""
... return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2., 5., 8.])

For a function that doesn't return a scalar, the number of dimensions in
`outarr` is the same as `arr`.

>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
 [3, 4, 9],
 [2, 5, 6]])

apply_over_axes(func, a, axes)
Apply a function repeatedly over multiple axes.

`func` is called as `res = func(a, axis)`, where `axis` is the first
element of `axes`. The result `res` of the function call must have
either the same dimensions as `a` or one less dimension. If `res`
has one less dimension than `a`, a dimension is inserted before
`axis`. The call to `func` is then repeated for each axis in `axes`,
with `res` as the first argument.

Parameters

func : function
 This function must take two arguments, `func(a, axis)`.

a : array_like
 Input array.

axes : array_like
 Axes over which `func` is applied; the elements must be integers.

>Returns

apply_over_axis : ndarray
 The output array. The number of dimensions is the same as `a`,
 but the shape can be different. This depends on whether `func`
 changes the shape of its output with respect to its input.

See Also

apply_along_axis :
 Apply a function to 1-D slices of an array along the given axis.

Notes

This function is equivalent to tuple axis arguments to reorderable ufuncs
with keepdims=True. Tuple axis arguments to ufuncs have been available since
version 1.7.0.

Examples

>>> a = np.arange(24).reshape(2,3,4)
>>> a
array([[[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]],
 [[12, 13, 14, 15]]])

```

```
[16, 17, 18, 19],
[20, 21, 22, 23]]))

Sum over axes 0 and 2. The result has same number of dimensions
as the original array:

>>> np.apply_over_axes(np.sum, a, [0,2])
array([[[60,
 [92,
 [124]]]])

Tuple axis arguments to ufuncs are equivalent:

>>> np.sum(a, axis=(0,2), keepdims=True)
array([[[60,
 [92,
 [124]]]])

arange(...)
arange([start, stop[, step[, dtype=None])

Return evenly spaced values within a given interval.

Values are generated within the half-open interval ``[start, stop)``
(in other words, the interval including `start` but excluding `stop`).
For integer arguments the function is equivalent to the Python built-in
`range <http://docs.python.org/lib/built-in-funcs.html>`_ function,
but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not
be consistent. It is better to use ``linspace`` for these cases.

Parameters

start : number, optional
 Start of interval. The interval includes this value. The default
 start value is 0.
stop : number
 End of interval. The interval does not include this value, except
 in some cases where `step` is not an integer and floating point
 round-off affects the length of `out`.
step : number, optional
 Spacing between values. For any output `out`, this is the distance
 between two adjacent values, ``out[i+1] - out[i]``. The default
 step size is 1. If `step` is specified, `start` must also be given.
dtype : dtype
 The type of the output array. If `dtype` is not given, infer the data
 type from the other input arguments.

Returns

arange : ndarray
 Array of evenly spaced values.

 For floating point arguments, the length of the result is
 ``ceil((stop - start)/step)``. Because of floating point overflow,
 this rule may result in the last element of `out` being greater
 than `stop`.

See Also

linspace: Evenly spaced numbers with careful handling of endpoints.
ogrid: Arrays of evenly spaced numbers in N-dimensions.
mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples

>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([0., 1., 2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])

argmax(a, axis=None, out=None)
 Returns the indices of the maximum values along an axis.

Parameters

a : array_like
 Input array.
axis : int, optional
 By default, the index is into the flattened array, otherwise
 along the specified axis.
out : array, optional
 If provided, the result will be inserted into this array. It should
```

be of the appropriate shape and [dtype](#).

Returns

-----

`index_array : ndarray` of ints  
Array of indices into the array. It has the same shape as `a.shape` with the dimension along `axis` removed.

See Also

-----

`ndarray.argmax`, `argmin`  
`amax` : The maximum value along a given axis.  
`unravel_index` : Convert a flat index into an index tuple.

Notes

-----

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

-----

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
 [3, 4, 5]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])

>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b) # Only the first occurrence is returned.
1
```

**argmin**(*a*, *axis=None*, *out=None*)

Returns the indices of the minimum values along an axis.

Parameters

-----

`a` : `array_like`  
Input array.

`axis` : `int`, optional  
By default, the index is into the flattened array, otherwise along the specified axis.

`out` : `array`, optional  
If provided, the result will be inserted into this array. It should be of the appropriate shape and [dtype](#).

Returns

-----

`index_array : ndarray` of ints  
Array of indices into the array. It has the same shape as `a.shape` with the dimension along `axis` removed.

See Also

-----

`ndarray.argmin`, `argmax`  
`amin` : The minimum value along a given axis.  
`unravel_index` : Convert a flat index into an index tuple.

Notes

-----

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

Examples

-----

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
 [3, 4, 5]])
>>> np.argmin(a)
0
>>> np.argmin(a, axis=0)
array([0, 0, 0])
>>> np.argmin(a, axis=1)
array([0, 0])

>>> b = np.arange(6)
>>> b[4] = 0
>>> b
array([0, 1, 2, 3, 0, 5])
>>> np.argmin(b) # Only the first occurrence is returned.
0
```

```
argpartition(a, kth, axis=-1, kind='introselect', order=None)
 Perform an indirect partition along the given axis using the algorithm
 specified by the `kind` keyword. It returns an array of indices of the
 same shape as `a` that index data along the given axis in partitioned
 order.

 .. versionadded:: 1.8.0

Parameters

a : array_like
 Array to sort.
kth : int or sequence of ints
 Element index to partition by. The kth element will be in its final
 sorted position and all smaller elements will be moved before it and
 all larger elements behind it.
 The order all elements in the partitions is undefined.
 If provided with a sequence of kth it will partition all of them into
 their sorted position at once.
axis : int or None, optional
 Axis along which to sort. The default is -1 (the last axis). If None,
 the flattened array is used.
kind : {'introselect'}, optional
 Selection algorithm. Default is 'introselect'
order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

Returns

index_array : ndarray, int
 Array of indices that partition `a` along the specified axis.
 In other words, ``a[index_array]`` yields a sorted `a`.

See Also

partition : Describes partition algorithms used.
ndarray.partition : Inplace partition.
argsort : Full indirect sort

Notes

See `partition` for notes on the different selection algorithms.

Examples

One dimensional array:

>>> x = np.array([3, 4, 2, 1])
>>> x[np.argpartition(x, 3)]
array([2, 1, 3, 4])
>>> x[np.argpartition(x, (1, 3))]
array([1, 2, 3, 4])

>>> x = [3, 4, 2, 1]
>>> np.array(x)[np.argpartition(x, 3)]
array([2, 1, 3, 4])

argsort(a, axis=-1, kind='quicksort', order=None)
 Returns the indices that would sort an array.

 Perform an indirect sort along the given axis using the algorithm specified
 by the `kind` keyword. It returns an array of indices of the same shape as
 `a` that index data along the given axis in sorted order.

Parameters

a : array_like
 Array to sort.
axis : int or None, optional
 Axis along which to sort. The default is -1 (the last axis). If None,
 the flattened array is used.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 Sorting algorithm.
order : str or list of str, optional
 When `a` is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
 be specified as a string, and not all fields need be specified,
 but unspecified fields will still be used, in the order in which
 they come up in the dtype, to break ties.

Returns

index_array : ndarray, int
 Array of indices that sort `a` along the specified axis.
 If `a` is one-dimensional, ``a[index_array]`` yields a sorted `a`.
```

See Also  
-----  
`sort` : Describes sorting algorithms used.  
`lexsort` : Indirect stable sort with multiple keys.  
`ndarray.sort` : Inplace sort.  
`argpartition` : Indirect partial sort.

Notes  
-----  
See `sort` for notes on the different sorting algorithms.  
As of NumPy 1.4.0 `argsort` works with real/[complex](#) arrays containing nan values. The enhanced sort order is documented in `sort`.

Examples  
-----  
One dimensional array:  
  
`>>> x = np.array([3, 1, 2])`  
`>>> np.argsort(x)`  
`array([1, 2, 0])`  
  
Two-dimensional array:  
  
`>>> x = np.array([[0, 3], [2, 2]])`  
`>>> x`  
`array([[0, 3],`  
 `[2, 2]])`  
  
`>>> np.argsort(x, axis=0)`  
`array([[0, 1],`  
 `[1, 0]])`  
  
`>>> np.argsort(x, axis=1)`  
`array([[0, 1],`  
 `[0, 1]])`  
  
Sorting with keys:  
  
`>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])`  
`>>> x`  
`array([(1, 0), (0, 1)],`  
 `dtype=[('x', '<i4'), ('y', '<i4')])`  
  
`>>> np.argsort(x, order=('x','y'))`  
`array([1, 0])`  
  
`>>> np.argsort(x, order=('y','x'))`  
`array([0, 1])`

**argwhere(a)**  
Find the indices of array elements that are non-zero, grouped by element.

Parameters  
-----  
`a : array_like`  
Input data.

Returns  
-----  
`index_array : ndarray`  
Indices of elements that are non-zero. Indices are grouped by element.

See Also  
-----  
`where`, `nonzero`

Notes  
-----  
``np.argwhere(a)`` is the same as ``np.transpose(np.nonzero(a))``.  
The output of ``argwhere`` is not suitable for indexing arrays.  
For this purpose use ``where(a)`` instead.

Examples  
-----  
`>>> x = np.arange(6).reshape(2,3)`  
`>>> x`  
`array([[0, 1, 2],`  
 `[3, 4, 5]])`  
`>>> np.argwhere(x>1)`  
`array([[0, 2],`  
 `[1, 0],`  
 `[1, 1],`  
 `[1, 2]])`

**around(a, decimals=0, out=None)**

```
Evenly round to the given number of decimals.

Parameters

a : array_like
 Input data.
decimals : int, optional
 Number of decimal places to round to (default: 0). If
 decimals is negative, it specifies the number of positions to
 the left of the decimal point.
out : ndarray, optional
 Alternative output array in which to place the result. It must have
 the same shape as the expected output, but the type of the output
 values will be cast if necessary. See `doc.ufuncs` (Section
 "Output arguments") for details.

Returns

rounded_array : ndarray
 An array of the same type as `a`, containing the rounded values.
 Unless `out` was specified, a new array is created. A reference to
 the result is returned.

 The real and imaginary parts of complex numbers are rounded
 separately. The result of rounding a float is a float.

See Also

ndarray.round : equivalent method

ceil, fix, floor, rint, trunc

Notes

For values exactly halfway between rounded decimal values, Numpy
rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0,
-0.5 and 0.5 round to 0.0, etc. Results may also be surprising due
to the inexact representation of decimal fractions in the IEEE
floating point standard [1]_ and errors introduced when scaling
by powers of ten.

References

... [1] "Lecture Notes on the Status of IEEE 754", William Kahan,

 http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

... [2] "How Futile are Mindless Assessments of
 Roundoff in Floating-Point Computation?", William Kahan,

 http://www.cs.berkeley.edu/~wkahan/MindLess.pdf

Examples

>>> np.around([0.37, 1.64])
array([0., 2.])
>>> np.around([0.37, 1.64], decimals=1)
array([0.4, 1.6])
>>> np.around([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value
array([0., 2., 2., 4., 4.])
>>> np.around([1,2,3,11], decimals=1) # ndarray of ints is returned
array([1, 2, 3, 11])
>>> np.around([1,2,3,11], decimals=-1)
array([0, 0, 0, 10])

array(...)
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)

Create an array.

Parameters

object : array_like
 An array, any object exposing the array interface, an
 object whose __array__ method returns an array, or any
 (nested) sequence.
dtype : data-type, optional
 The desired data-type for the array. If not given, then
 the type will be determined as the minimum type required
 to hold the objects in the sequence. This argument can only
 be used to 'upcast' the array. For downcasting, use the
 .astype\(t\) method.
copy : bool, optional
 If true (default), then the object is copied. Otherwise, a copy
 will only be made if __array__ returns a copy, if obj is a
 nested sequence, or if a copy is needed to satisfy any of the other
 requirements ('dtype', 'order', etc.).
order : {'C', 'F', 'A'}, optional
 Specify the order of the array. If order is 'C', then the array
 will be in C-contiguous order (last-index varies the fastest).
 If order is 'F', then the returned array will be in
```

```

Fortran-contiguous order (first-index varies the fastest).
If order is 'A' (default), then the returned array may be
in any order (either C-, Fortran-contiguous, or even discontiguous),
unless a copy is required, in which case it will be C-contiguous.
subok : bool, optional
 If True, then sub-classes will be passed-through, otherwise
 the returned array will be forced to be a base-class array (default).
ndmin : int, optional
 Specifies the minimum number of dimensions that the resulting
 array should have. Ones will be pre-pended to the shape as
 needed to meet this requirement.

>Returns

out : ndarray
 An array object satisfying the specified requirements.

See Also

empty, empty_like, zeros, zeros_like, ones, ones_like, fill

Examples

>>> np.array([1, 2, 3])
array([1, 2, 3])

Upcasting:
>>> np.array([1, 2, 3.0])
array([1., 2., 3.])

More than one dimension:
>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
 [3, 4]])

Minimum dimensions 2:
>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])

Type provided:
>>> np.array([1, 2, 3], dtype=complex)
array([1.+0.j, 2.+0.j, 3.+0.j])

Data-type consisting of more than one element:
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])

Creating an array from sub-classes:
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
 [3, 4]])

>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
 [3, 4]])

array2string(a, max_line_width=None, precision=None, suppress_small=None, separator=' ', prefix='', style=
<built-in function repr>, formatter=None)
 Return a string representation of an array.

Parameters

a : ndarray
 Input array.
max_line_width : int, optional
 The maximum number of columns the string should span. Newline
 characters splits the string appropriately after array elements.
precision : int, optional
 Floating point precision. Default is the current printing
 precision (usually 8), which can be altered using `set_printoptions`.
suppress_small : bool, optional
 Represent very small numbers as zero. A number is "very small" if it
 is smaller than the current printing precision.
separator : str, optional
 Inserted between elements.
prefix : str, optional
 An array is typically printed as::
 'prefix(' + array2string(a) + ')'

 The length of the prefix string is used to align the
 output correctly.

```

```
style : function, optional
 A function that accepts an ndarray and returns a string. Used only
 when the shape of `a` is equal to ``()`` , i.e. for 0-D arrays.
formatter : dict of callables, optional
 If not None, the keys should indicate the type(s) that the respective
 formatting function applies to. Callables should return a string.
 Types that are not specified (by their corresponding keys) are handled
 by the default formatters. Individual types for which a formatter
 can be set are::
 - 'bool'
 - 'int'
 - 'timedelta' : a `numpy.timedelta64``
 - 'datetime' : a `numpy.datetime64``
 - 'float'
 - 'longfloat' : 128-bit floats
 - 'complexfloat'
 - 'longcomplexfloat' : composed of two 128-bit floats
 - 'numpy_str' : types `numpy.string` and `numpy.unicode`_
 - 'str' : all other strings

Other keys that can be used to set a group of types at once are::
 - 'all' : sets all types
 - 'int_kind' : sets 'int'
 - 'float_kind' : sets 'float' and 'longfloat'
 - 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
 - 'str_kind' : sets 'str' and 'numpystr'

Returns

array_str : str
 String representation of the array.

Raises

TypeError
 if a callable in `formatter` does not return a string.

See Also

array_str, array_repr, set_printoptions, get_printoptions

Notes

If a formatter is specified for a certain type, the 'precision' keyword is
ignored for that type.

This is a very flexible function; `array_repr` and `array_str` are using
`array2string` internally so keywords with the same name should work
identically in all three functions.

Examples

>>> x = np.array\\\\(\\\\[1e-16,1,2,3\\\\]\\\\)
>>> print\\\\(np.array2string\\\\(x, precision=2, separator=',',
... suppress_small=True\\\\)\\\\)
\\\\[0., 1., 2., 3.\\\\]
>>> x = np.arange\\\\(3.\\\\)
>>> np.array2string\\\\(x, formatter={'float_kind':lambda x: "%2f" % x}\\\\)
'\\\\[0.00 1.00 2.00\\\\]'
>>> x = np.arange\\\\(3\\\\)
>>> np.array2string\\\\(x, formatter={'int':lambda x: hex\\\\(x\\\\)}\\\\)
'\\\\[0x0L 0x1L 0x2L\\\\]'

array_equal\\\\(a1, a2\\\\)
True if two arrays have the same shape and elements, False otherwise.

Parameters

a1, a2 : array_like
 Input arrays.

Returns

b : bool
 Returns True if the arrays are equal.

See Also

allclose: Returns True if two arrays are element-wise equal within a
 tolerance.
array_equiv: Returns True if input arrays are shape consistent and all
 elements equal.

Examples

```

```

>>> np.array_equal([1, 2], [1, 2])
True
>>> np.array_equal(np.array([1, 2]), np.array([1, 2]))
True
>>> np.array_equal([1, 2], [1, 2, 3])
False
>>> np.array_equal([1, 2], [1, 4])
False

array_equiv(a1, a2)
 Returns True if input arrays are shape consistent and all elements equal.

 Shape consistent means they are either the same shape, or one input array
 can be broadcasted to create the same shape as the other one.

 Parameters

 a1, a2 : array_like
 Input arrays.

 Returns

 out : bool
 True if equivalent, False otherwise.

 Examples

 >>> np.array_equiv([1, 2], [1, 2])
True
>>> np.array_equiv([1, 2], [1, 3])
False

 Showing the shape equivalence:

 >>> np.array_equiv([1, 2], [[1, 2], [1, 2]])
True
>>> np.array_equiv([1, 2], [[1, 2, 1, 2], [1, 2, 1, 2]])
False

 >>> np.array_equiv([1, 2], [[1, 2], [1, 3]])
False

array_repr(arr, max_line_width=None, precision=None, suppress_small=None)
 Return the string representation of an array.

 Parameters

 arr : ndarray
 Input array.
 max_line_width : int, optional
 The maximum number of columns the string should span. Newline
 characters split the string appropriately after array elements.
 precision : int, optional
 Floating point precision. Default is the current printing precision
 (usually 8), which can be altered using `set_printoptions`.
 suppress_small : bool, optional
 Represent very small numbers as zero, default is False. Very small
 is defined by `precision`, if the precision is 8 then
 numbers smaller than 5e-9 are represented as zero.

 Returns

 string : str
 The string representation of an array.

 See Also

 array_str, array2string, set_printoptions

 Examples

 >>> np.array_repr(np.array([1,2]))
'array([1, 2])'
>>> np.array_repr(np.ma.array([0.]))
'MaskedArray([0.])'
>>> np.array_repr(np.array([], np.int32))
'array([], dtype=int32)'

 >>> x = np.array([1e-6, 4e-7, 2, 3])
 >>> np.array_repr(x, precision=6, suppress_small=True)
'array([0.000001, 0. , 2. , 3.])'

array_split(ary, indices_or_sections, axis=0)
 Split an array into multiple sub-arrays.

 Please refer to the ``split`` documentation. The only difference
 between these functions is that ``array_split`` allows
 `indices_or_sections` to be an integer that does *not* equally

```

divide the axis.

See Also

-----

`split` : Split array into multiple sub-arrays of equal size.

Examples

-----

```
>>> x = np.arange(8.0)
>>> np.array_split(x, 3)
[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7.])]
```

**array\_str(a, max\_line\_width=None, precision=None, suppress\_small=None)**

Return a string representation of the data in an array.

The data in the array is returned as a `single` string. This function is similar to `array\_repr`, the difference being that `array\_repr` also returns information on the kind of array and its data type.

Parameters

-----

`a` : `ndarray`  
Input array.

`max_line_width` : `int`, optional  
Inserts newlines if text is longer than `max\_line\_width`. The default is, indirectly, 75.

`precision` : `int`, optional  
Floating point precision. Default is the current printing precision (usually 8), which can be altered using `set\_printoptions`.

`suppress_small` : `bool`, optional  
Represent numbers "very close" to zero as zero; default is False. Very close is defined by precision: if the precision is 8, e.g., numbers smaller (in absolute value) than 5e-9 are represented as zero.

See Also

-----

`array2string`, `array_repr`, `set_printoptions`

Examples

-----

```
>>> np.array_str(np.arange(3))
'[0 1 2]'
```

**asanyarray(a, dtype=None, order=None)**

Convert the input to an `ndarray`, but pass `ndarray` subclasses through.

Parameters

-----

`a` : `array_like`  
Input data, in any form that can be converted to an array. This includes scalars, lists, lists of tuples, tuples, tuples of tuples, tuples of lists, and ndarrays.

`dtype` : data-type, optional  
By default, the data-type is inferred from the input data.

`order` : {'C', 'F'}, optional  
Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

Returns

-----

`out` : `ndarray` or an `ndarray` subclass  
Array interpretation of `a`. If `a` is an `ndarray` or a subclass of `ndarray`, it is returned as-is and no copy is performed.

See Also

-----

`asarray` : Similar function which always returns ndarrays.  
`ascontiguousarray` : Convert input to a contiguous array.  
`asfarray` : Convert input to a `Floating` point `ndarray`.  
`asfortranarray` : Convert input to an `ndarray` with column-major memory order.  
`asarray_chkfinite` : Similar function which checks input for NaNs and Infs.  
`fromiter` : Create an array from an iterator.  
`fromfunction` : Construct an array by executing a function on grid positions.

Examples

-----

Convert a list into an array:

```
>>> a = [1, 2]
>>> np.asanyarray(a)
array([1, 2])
```

Instances of `ndarray` subclasses are passed through as-is:

```
>>> a = np.matrix([1, 2])
>>> np.asanyarray(a) is a
True

asarray(a, dtype=None, order=None)
Convert the input to an array.

Parameters

a : array_like
 Input data, in any form that can be converted to an array. This
 includes lists, lists of tuples, tuples, tuples of tuples, tuples
 of lists and ndarrays.
dtype : data-type, optional
 By default, the data-type is inferred from the input data.
order : {'C', 'F'}, optional
 Whether to use row-major (C-style) or
 column-major (Fortran-style) memory representation.
 Defaults to 'C'.

Returns

out : ndarray
 Array interpretation of `a`. No copy is performed if the input
 is already an ndarray. If `a` is a subclass of ndarray, a base
 class ndarray is returned.

See Also

asanyarray : Similar function which passes through subclasses.
ascontiguousarray : Convert input to a contiguous array.
asfarray : Convert input to a floating point ndarray.
asfortranarray : Convert input to an ndarray with column-major
 memory order.
asarray_chkfinite : Similar function which checks input for NaNs and Infs.
fromiter : Create an array from an iterator.
fromfunction : Construct an array by executing a function on grid
 positions.

Examples

Convert a list into an array:

>>> a = [1, 2]
>>> np.asarray(a)
array([1, 2])

Existing arrays are not copied:

>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True

If `dtype` is set, array is copied only if `dtype` does not match:

>>> a = np.array([1, 2], dtype=np.float32)
>>> np.asarray(a, dtype=np.float32) is a
True
>>> np.asarray(a, dtype=np.float64) is a
False

Contrary to `asanyarray`, ndarray subclasses are not passed through:

>>> issubclass(np.matrix, np.ndarray)
True
>>> a = np.matrix([[1, 2]])
>>> np.asarray(a) is a
False
>>> np.asanyarray(a) is a
True

asarray_chkfinite(a, dtype=None, order=None)
Convert the input to an array, checking for NaNs or Infs.

Parameters

a : array_like
 Input data, in any form that can be converted to an array. This
 includes lists, lists of tuples, tuples, tuples of tuples, tuples
 of lists and ndarrays. Success requires no NaNs or Infs.
dtype : data-type, optional
 By default, the data-type is inferred from the input data.
order : {'C', 'F'}, optional
 Whether to use row-major (C-style) or
 column-major (Fortran-style) memory representation.
 Defaults to 'C'.
```

Returns

```

out : ndarray
 Array interpretation of `a`. No copy is performed if the input
 is already an ndarray. If `a` is a subclass of ndarray, a base
 class ndarray is returned.

Raises

ValueError
 Raises ValueError if `a` contains NaN (Not a Number) or Inf (Infinity).

See Also

asarray : Create an array.
asanyarray : Similar function which passes through subclasses.
ascontiguousarray : Convert input to a contiguous array.
asfarray : Convert input to a floating point ndarray.
asfortranarray : Convert input to an ndarray with column-major
 memory order.
fromiter : Create an array from an iterator.
fromfunction : Construct an array by executing a function on grid
 positions.

Examples

Convert a list into an array. If all elements are finite
``asarray_chkfinite`` is identical to ``asarray``.

->>> a = [1, 2]
->>> np.asarray_chkfinite(a, dtype=float)
array([1., 2.])

Raises ValueError if array_like contains Nans or Infs.

->>> a = [1, 2, np.inf]
->>> try:
... np.asarray_chkfinite(a)
... except ValueError:
... print('ValueError')
...
ValueError

ascontiguousarray(a, dtype=None)
Return a contiguous array in memory (C order).

Parameters

a : array_like
 Input array.
dtype : str or dtype object, optional
 Data-type of returned array.

Returns

out : ndarray
 Contiguous array of same shape and content as `a`, with type `dtype`
 if specified.

See Also

asfortranarray : Convert input to an ndarray with column-major
 memory order.
require : Return an ndarray that satisfies requirements.
ndarray.flags : Information about the memory layout of the array.

Examples

->>> x = np.arange(6).reshape(2,3)
->>> np.ascontiguousarray(x, dtype=np.float32)
array([[0., 1., 2.],
 [3., 4., 5.]], dtype=float32)
->>> x.flags['C_CONTIGUOUS']
True

asfarray(a, dtype=<type 'numpy.float64'>)
Return an array converted to a float type.

Parameters

a : array_like
 The input array.
dtype : str or dtype object, optional
 Float type code to coerce input array `a`. If `dtype` is one of the
 'int' dtypes, it is replaced with float64.

Returns

out : ndarray
```

```
The input `a` as a float ndarray.
Examples

>>> np.asfarray([2, 3])
array([2., 3.])
>>> np.asfarray([2, 3], dtype='float')
array([2., 3.])
>>> np.asfarray([2, 3], dtype='int8')
array([2., 3.])

asfortranarray(a, dtype=None)
Return an array laid out in Fortran order in memory.

Parameters

a : array_like
 Input array.
dtype : str or dtype object, optional
 By default, the data-type is inferred from the input data.

Returns

out : ndarray
 The input `a` in Fortran, or column-major, order.

See Also

ascontiguousarray : Convert input to a contiguous (C order) array.
asanyarray : Convert input to an ndarray with either row or
 column-major memory order.
require : Return an ndarray that satisfies requirements.
ndarray.flags : Information about the memory layout of the array.

Examples

>>> x = np.arange(6).reshape(2,3)
>>> y = np.asfortranarray(x)
>>> x.flags['F_CONTIGUOUS']
False
>>> y.flags['F_CONTIGUOUS']
True

asmatrix(data, dtype=None)
Interpret the input as a matrix.

Unlike `matrix`, `asmatrix` does not make a copy if the input is already
a matrix or an ndarray. Equivalent to ``matrix(data, copy=False)``.

Parameters

data : array_like
 Input data.
dtype : data-type
 Data-type of the output matrix.

Returns

mat : matrix
 `data` interpreted as a matrix.

Examples

>>> x = np.array([[1, 2], [3, 4]])
>>> m = np.asmatrix(x)
>>> x[0,0] = 5

>>> m
matrix([[5, 2],
 [3, 4]])

asscalar(a)
Convert an array of size 1 to its scalar equivalent.

Parameters

a : ndarray
 Input array of size 1.

Returns

out : scalar
 Scalar representation of `a`. The output data type is the same type
 returned by the input's `item` method.

Examples
```

```

>>> np.asscalar(np.array([24]))
24

atleast_1d(*arys)
 Convert inputs to arrays with at least one dimension.

 Scalar inputs are converted to 1-dimensional arrays, whilst
 higher-dimensional inputs are preserved.

 Parameters

 arys1, arys2, ... : array_like
 One or more input arrays.

 Returns

 ret : ndarray
 An array, or sequence of arrays, each with ``a.ndim >= 1``.
 Copies are made only if necessary.

 See Also

 atleast_2d, atleast_3d

 Examples

 >>> np.atleast_1d(1.0)
 array([1.])

 >>> x = np.arange(9.0).reshape(3,3)
 >>> np.atleast_1d(x)
 array([[0., 1., 2.],
 [3., 4., 5.],
 [6., 7., 8.]])
 >>> np.atleast_1d(x) is x
 True

 >>> np.atleast_1d(1, [3, 4])
 [array([1]), array([3, 4])]

atleast_2d(*arys)
 View inputs as arrays with at least two dimensions.

 Parameters

 arys1, arys2, ... : array_like
 One or more array-like sequences. Non-array inputs are converted
 to arrays. Arrays that already have two or more dimensions are
 preserved.

 Returns

 res, res2, ... : ndarray
 An array, or tuple of arrays, each with ``a.ndim >= 2``.
 Copies are avoided where possible, and views with two or more
 dimensions are returned.

 See Also

 atleast_1d, atleast_3d

 Examples

 >>> np.atleast_2d(3.0)
 array([[3.]])

 >>> x = np.arange(3.0)
 >>> np.atleast_2d(x)
 array([[0., 1., 2.]])
 >>> np.atleast_2d(x).base is x
 True

 >>> np.atleast_2d(1, [1, 2], [[1, 2]])
 [array([[1]]), array([[1, 2]]), array([[1, 2]])]

atleast_3d(*arys)
 View inputs as arrays with at least three dimensions.

 Parameters

 arys1, arys2, ... : array_like
 One or more array-like sequences. Non-array inputs are converted to
 arrays. Arrays that already have three or more dimensions are
 preserved.

 Returns

```

```
res1, res2, ... : ndarray
 An array, or tuple of arrays, each with ``a.ndim >= 3``. Copies are
 avoided where possible, and views with three or more dimensions are
 returned. For example, a 1-D array of shape ``(N,)`` becomes a view
 of shape ``(1, N, 1)``, and a 2-D array of shape ``(M, N)`` becomes a
 view of shape ``(M, N, 1)``.

See Also

atleast_1d, atleast_2d

Examples

>>> np.atleast_3d(3.0)
array([[[3.]]])

>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)

>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x
True

>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
... print(arr, arr.shape)
...
[[[1]
 [2]]] (1, 2, 1)
[[[1]
 [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)

average(a, axis=None, weights=None, returned=False)
Compute the weighted average along the specified axis.

Parameters

a : array_like
 Array containing data to be averaged. If `a` is not an array, a
 conversion is attempted.
axis : int, optional
 Axis along which to average `a`. If `None`, averaging is done over
 the flattened array.
weights : array_like, optional
 An array of weights associated with the values in `a`. Each value in
 `a` contributes to the average according to its associated weight.
 The weights array can either be 1-D (in which case its length must be
 the size of `a` along the given axis) or of the same shape as `a`.
 If `weights=None`, then all data in `a` are assumed to have a
 weight equal to one.
returned : bool, optional
 Default is `False`. If `True`, the tuple (`average`, `sum_of_weights`)
 is returned, otherwise only the average is returned.
 If `weights=None`, `sum_of_weights` is equivalent to the number of
 elements over which the average is taken.

Returns

average, [sum_of_weights] : array_type or double
 Return the average along the specified axis. When returned is `True`,
 return a tuple with the average as the first element and the sum
 of the weights as the second element. The return type is `Float`
 if `a` is of integer type, otherwise it is of the same type as `a`.
 `sum_of_weights` is of the same type as `average`.

Raises

ZeroDivisionError
 When all weights along axis are zero. See `numpy.ma.average` for a
 version robust to this type of error.
TypeError
 When the length of 1D `weights` is not the same as the shape of `a`
 along axis.

See Also

mean

ma.average : average for masked arrays -- useful if your data contains
 "missing" values

Examples

>>> data = range(1,5)
>>> data
```

```
[1, 2, 3, 4]
>>> np.average(data)
2.5
>>> np.average(range(1,11), weights=range(10,0,-1))
4.0

>>> data = np.arange(6).reshape((3,2))
>>> data
array([[0, 1],
 [2, 3],
 [4, 5]])
>>> np.average(data, axis=1, weights=[1./4, 3./4])
array([0.75, 2.75, 4.75])
>>> np.average(data, weights=[1./4, 3./4])
Traceback (most recent call last):
...
TypeError: Axis must be specified when shapes of a and weights differ.
```

**bartlett(M)**

Return the Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

## Parameters

M : [int](#)  
     Number of points in the output window. If zero or less, an empty array is returned.

Returns

out : [array](#)  
     The triangular window, with the maximum value normalized to one (the value one appears only if the [number](#) of samples is odd), with the first and last samples equal to zero.

## See Also

[blackman](#), [hamming](#), [hanning](#), [kaiser](#)

## Notes

The Bartlett window is defined as

$$\dots \text{math:: } w(n) = \frac{2}{M-1} \left[ \frac{M-1}{2} - \left| n - \frac{M-1}{2} \right| \right]$$

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. The Fourier transform of the Bartlett is the product of two sinc functions.

Note the excellent discussion in Kanasewich.

## References

- .. [1] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", *Biometrika* 37, 1-16, 1950.
- .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", *The University of Alberta Press*, 1975, pp. 109-110.
- .. [3] A.V. Oppenheim and R.W. Schafer, "Discrete-Time Signal Processing", Prentice-Hall, 1999, pp. 468-471.
- .. [4] Wikipedia, "Window function", [http://en.wikipedia.org/wiki/Window\\_function](http://en.wikipedia.org/wiki/Window_function)
- .. [5] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 429.

## Examples

```
>>> np.bartlett(12)
array([0. , 0.18181818, 0.36363636, 0.54545455, 0.72727273,
 0.90909091, 0.90909091, 0.72727273, 0.54545455, 0.36363636,
 0.18181818, 0.])
```

Plot the window and its frequency response (requires SciPy and matplotlib):

```
>>> from numpy.fft import fft, fftshift
>>> window = np.bartlett(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>
>>> plt.title("Bartlett window")
```

```

<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Bartlett window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()

```

**base\_repr**(number, base=2, padding=0)

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

## Parameters

`number` : `int`  
     The value to convert. Only positive values are handled.  
`base` : `int`, optional  
     Convert `'number'` to the `base` `number` system. The valid range is 2-36,  
     the default value is 2.  
`padding` : `int`, optional  
     Number of zeros padded on the left. Default is 0 (no padding).

## Returns

`out` : `str`  
     String representation of `'number'` in `base` system.

## See Also

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

## Examples

```

>>> np.base_repr(5)
'101'
>>> np.base_repr(6, 5)
'11'
>>> np.base_repr(7, base=5, padding=3)
'00012'

>>> np.base_repr(10, base=16)
'A'
>>> np.base_repr(32, base=16)
'20'

```

**binary\_repr**(num, width=None)

Return the binary representation of the input `number` as a string.

For negative numbers, if `width` is not given, a minus sign is added to the front. If `width` is given, the two's complement of the `number` is returned, with respect to that `width`.

In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two's-complement system can represent every `integer` in the range `:math:`-2^{N-1}`` to `:math:`+2^{N-1}-1``.

## Parameters

`num` : `int`  
     Only an `integer` decimal `number` can be used.  
`width` : `int`, optional  
     The length of the returned string if `'num'` is positive, the length of  
     the two's complement if `'num'` is negative.

## Returns

`bin` : `str`  
     Binary representation of `'num'` or two's complement of `'num'`.

See Also

-----

`base_repr`: Return a string representation of a [number](#) in the given base system.

Notes

-----

`'binary_repr'` is equivalent to using `'base_repr'` with base 2, but about 25x faster.

References

-----

.. [1] Wikipedia, "Two's complement",  
[http://en.wikipedia.org/wiki/Two's\\_complement](http://en.wikipedia.org/wiki/Two's_complement)

Examples

-----

```
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

The two's complement is returned when the input [number](#) is negative and width is specified:

```
>>> np.binary_repr(-3, width=4)
'1101'
```

**bincount(...)**

`bincount`(*x*, weights=None, minlength=None)

Count [number](#) of occurrences of each value in array of non-negative ints.

The [number](#) of bins (of size 1) is one larger than the largest value in '*x*'. If 'minlength' is specified, there will be at least this [number](#) of bins in the output array (though it will be longer if necessary, depending on the contents of '*x*').

Each bin gives the [number](#) of occurrences of its index value in '*x*'. If 'weights' is specified the input array is weighted by it, i.e. if a value ``*n*`` is found at position ``*i*``, ``*out*[*n*] += weight[i]`` instead of ``*out*[*n*] += 1``.

Parameters

-----

```
x : array_like, 1 dimension, nonnegative ints
 Input array.
weights : array_like, optional
 Weights, array of the same shape as 'x'.
minlength : int, optional
 A minimum number of bins for the output array.
```

.. versionadded:: 1.6.0

Returns

-----

```
out : ndarray of ints
 The result of binning the input array.
 The length of 'out' is equal to ``np.amax(x)+1``.
```

Raises

-----

**ValueError**

If the input is not 1-dimensional, or contains elements with negative values, or if 'minlength' is non-positive.

**TypeError**

If the type of the input is [float](#) or [complex](#).

See Also

-----

`histogram`, `digitize`, `unique`

Examples

-----

```
>>> np.bincount(np.arange(5))
array([1, 1, 1, 1, 1])
>>> np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
array([1, 3, 1, 1, 0, 0, 0, 1])

>>> x = np.array([0, 1, 1, 3, 2, 1, 7, 23])
>>> np.bincount(x).size == np.amax(x)+1
True
```

The input array needs to be of [integer dtype](#), otherwise a `TypeError` is raised:

```
>>> np.bincount(np.arange(5, dtype=np.float))
Traceback (most recent call last):
```

```

 File "<stdin>", line 1, in <module>
TypeError: array cannot be safely cast to required type

A possible use of ``bincount`` is to perform sums over
variable-size chunks of an array, using the ``weights`` keyword.

>>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6]) # weights
>>> x = np.array([0, 1, 1, 2, 2, 2])
>>> np.bincount(x, weights=w)
array([0.3, 0.7, 1.1])

```

**blackman(M)**

Return the Blackman window.

The Blackman window is a taper formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a Kaiser window.

Parameters

M : `int`  
Number of points in the output window. If zero or less, an empty array is returned.

Returns

out : `ndarray`  
The window, with the maximum value normalized to one (the value one appears only if the `number` of samples is odd).

See Also

bartlett, hamming, hanning, kaiser

Notes

The Blackman window is defined as

$$\dots \text{math::: } w(n) = 0.42 - 0.5 \cos(2\pi n/M) + 0.08 \cos(4\pi n/M)$$

Most references to the Blackman window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. It is known as a "near optimal" tapering function, almost as good (by some measures) as the kaiser window.

References

Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

Oppenheim, A.V., and R.W. Schafer. Discrete-Time Signal Processing. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

Examples

```

>>> np.blackman(12)
array([-1.38777878e-17, 3.26064346e-02, 1.59903635e-01,
 4.14397981e-01, 7.36045180e-01, 9.67046769e-01,
 9.67046769e-01, 7.36045180e-01, 4.14397981e-01,
 1.59903635e-01, 3.26064346e-02, -1.38777878e-17])

```

Plot the window and the frequency response:

```

>>> from numpy.fft import fft, fftshift
>>> window = np.blackman(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Blackman window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]

```

```
>>> plt.title("Frequency response of Blackman window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()

bmat(obj, ldict=None, gdict=None)
Build a matrix object from a string, nested sequence, or array.

Parameters

obj : str or array_like
 Input data. Names of variables in the current scope may be
 referenced, even if `obj` is a string.
ldict : dict, optional
 A dictionary that replaces local operands in current frame.
 Ignored if `obj` is not a string or `gdict` is `None`.
gdict : dict, optional
 A dictionary that replaces global operands in current frame.
 Ignored if `obj` is not a string.

Returns

out : matrix
 Returns a matrix object, which is a specialized 2-D array.

See Also

matrix

Examples

>>> A = np.mat('1 1; 1 1')
>>> B = np.mat('2 2; 2 2')
>>> C = np.mat('3 4; 5 6')
>>> D = np.mat('7 8; 9 0')

All the following expressions construct the same block matrix:
>>> np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
 [1, 1, 2, 2],
 [3, 4, 7, 8],
 [5, 6, 9, 0]])
>>> np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
matrix([[1, 1, 2, 2],
 [1, 1, 2, 2],
 [3, 4, 7, 8],
 [5, 6, 9, 0]])
>>> np.bmat('A,B; C,D')
matrix([[1, 1, 2, 2],
 [1, 1, 2, 2],
 [3, 4, 7, 8],
 [5, 6, 9, 0]])

broadcast_arrays(*args, **kwargs)
Broadcast any number of arrays against each other.

Parameters

`*args` : array_likes
 The arrays to broadcast.
subok : bool, optional
 If True, then sub-classes will be passed-through, otherwise
 the returned arrays will be forced to be a base-class array (default).

Returns

broadcasted : list of arrays
 These arrays are views on the original arrays. They are typically
 not contiguous. Furthermore, more than one element of a
 broadcasted array may refer to a single memory location. If you
 need to write to the arrays, make copies first.

Examples

>>> x = np.array([[1,2,3]])
>>> y = np.array([[1],[2],[3]])
>>> np.broadcast_arrays(x, y)
array([[1, 2, 3],
 [1, 2, 3],
 [1, 2, 3]]), array([[[1, 1, 1],
 [2, 2, 2]],
```

```
[3, 3, 3]]])
Here is a useful idiom for getting contiguous copies instead of
non-contiguous views.
>>> [np.array(a) for a in np.broadcast_arrays(x, y)]
[array([[1, 2, 3],
 [1, 2, 3]]), array([[1, 1, 1],
 [2, 2, 2],
 [3, 3, 3]])]
broadcast_to(array, shape, subok=False)
Broadcast an array to a new shape.

Parameters

array : array_like
 The array to broadcast.
shape : tuple
 The shape of the desired array.
subok : bool, optional
 If True, then sub-classes will be passed-through, otherwise
 the returned array will be forced to be a base-class array (default).

Returns

broadcast : array
 A readonly view on the original array with the given shape. It is
 typically not contiguous. Furthermore, more than one element of a
 broadcasted array may refer to a single memory location.

Raises

ValueError
 If the array is not compatible with the new shape according to NumPy's
 broadcasting rules.

Notes

.. versionadded:: 1.10.0

Examples

>>> x = np.array([1, 2, 3])
>>> np.broadcast_to(x, (3, 3))
array([[1, 2, 3],
 [1, 2, 3],
 [1, 2, 3]])

busday_count(...)
busday_count(begindates, enddates, weekmask='1111100', holidays=[], busdaycal=None, out=None)
Counts the number of valid days between `begindates` and
'enddates', not including the day of `enddates'.
If ``enddates`` specifies a date value that is earlier than the
corresponding ``begindates`` date value, the count will be negative.
.. versionadded:: 1.7.0

Parameters

begindates : array_like of datetime64\[D\]
 The array of the first dates for counting.
enddates : array_like of datetime64\[D\]
 The array of the end dates for counting, which are excluded
 from the count themselves.
weekmask : str or array_like of bool, optional
 A seven-element array indicating which of Monday through Sunday are
 valid days. May be specified as a length-seven list or array, like
 [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string
 like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for
 weekdays, optionally separated by white space. Valid abbreviations
 are: Mon Tue Wed Thu Fri Sat Sun
holidays : array_like of datetime64\[D\], optional
 An array of dates to consider as invalid dates. They may be
 specified in any order, and NaT (not-a-time) dates are ignored.
 This list is saved in a normalized form that is suited for
 fast calculations of valid days.
busdaycal : busdaycalendar, optional
 A `busdaycalendar` object which specifies the valid days. If this
 parameter is provided, neither weekmask nor holidays may be
 provided.
out : array of int, optional
 If provided, this array is filled with the result.

Returns
```

```

out : array of int
 An array with a shape from broadcasting ``begindates`` and ``enddates``
 together, containing the number of valid days between
 the begin and end dates.

See Also

busdaycalendar: An object that specifies a custom set of valid days.
is_busday : Returns a boolean array indicating valid days.
busday_offset : Applies an offset counted in valid days.

Examples

>>> # Number of weekdays in January 2011
... np.busday_count('2011-01', '2011-02')
21
>>> # Number of weekdays in 2011
... np.busday_count('2011', '2012')
260
>>> # Number of Saturdays in 2011
... np.busday_count('2011', '2012', weekmask='Sat')
53

busday_offset(...)
busday_offset(dates, offsets, roll='raise', weekmask='1111100', holidays=None, busdaycal=None, out=None)

First adjusts the date to fall on a valid day according to
the ``roll`` rule, then applies offsets to the given dates
counted in valid days.

.. versionadded:: 1.7.0

Parameters

dates : array_like of datetime64\[D\]
 The array of dates to process.
offsets : array_like of int
 The array of offsets, which is broadcast with ``dates``.
roll : {'raise', 'nat', 'forward', 'following', 'backward', 'preceding', 'modifiedfollowing', 'modifiedpreceding'}, optional
 How to treat dates that do not fall on a valid day. The default
 is 'raise'.
 * 'raise' means to raise an exception for an invalid day.
 * 'nat' means to return a NaT (not-a-time) for an invalid day.
 * 'forward' and 'following' mean to take the first valid day
 later in time.
 * 'backward' and 'preceding' mean to take the first valid day
 earlier in time.
 * 'modifiedfollowing' means to take the first valid day
 later in time unless it is across a Month boundary, in which
 case to take the first valid day earlier in time.
 * 'modifiedpreceding' means to take the first valid day
 earlier in time unless it is across a Month boundary, in which
 case to take the first valid day later in time.
weekmask : str or array_like of bool, optional
 A seven-element array indicating which of Monday through Sunday are
 valid days. May be specified as a length-seven list or array, like
 [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string
 like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for
 weekdays, optionally separated by white space. Valid abbreviations
 are: Mon Tue Wed Thu Fri Sat Sun
holidays : array_like of datetime64\[D\], optional
 An array of dates to consider as invalid dates. They may be
 specified in any order, and NaT (not-a-time) dates are ignored.
 This list is saved in a normalized form that is suited for
 fast calculations of valid days.
busdaycal : busdaycalendar, optional
 A busdaycalendar object which specifies the valid days. If this
 parameter is provided, neither weekmask nor holidays may be
 provided.
out : array of datetime64\[D\], optional
 If provided, this array is filled with the result.

Returns

out : array of datetime64\[D\]
 An array with a shape from broadcasting ``dates`` and ``offsets``
 together, containing the dates with offsets applied.

See Also

busdaycalendar: An object that specifies a custom set of valid days.
is_busday : Returns a boolean array indicating valid days.
busday_count : Counts how many valid days are in a half-open date range.

Examples

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

```
... np.busday_offset('2011-10', 0, roll='forward')
numpy.datetime64('2011-10-03','D')
>>> # Last business day in February 2012 (not accounting for holidays)
... np.busday_offset('2012-03', -1, roll='forward')
numpy.datetime64('2012-02-29','D')
>>> # Third Wednesday in January 2011
... np.busday_offset('2011-01', 2, roll='forward', weekmask='Wed')
numpy.datetime64('2011-01-19','D')
>>> # 2012 Mother's Day in Canada and the U.S.
... np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')
numpy.datetime64('2012-05-13','D')

>>> # First business day on or after a date
... np.busday_offset('2011-03-20', 0, roll='forward')
numpy.datetime64('2011-03-21','D')
>>> np.busday_offset('2011-03-22', 0, roll='forward')
numpy.datetime64('2011-03-22','D')
>>> # First business day after a date
... np.busday_offset('2011-03-20', 1, roll='backward')
numpy.datetime64('2011-03-21','D')
>>> np.busday_offset('2011-03-22', 1, roll='backward')
numpy.datetime64('2011-03-23','D')
```

**byte\_bounds(a)**

Returns pointers to the end-points of an array.

## Parameters

a : `ndarray`  
Input array. It must conform to the Python-side of the array interface.

## Returns

(low, high) : tuple of 2 integers  
The first `integer` is the first `byte` of the array, the second `integer` is just past the last `byte` of the array. If `a` is not contiguous it will not use every `byte` between the (`low`, `high`) values.

## Examples

```
>>> I = np.eve(2, dtype='f'); I.dtype
dtype('float32')
>>> low, high = np.byte_bounds(I)
>>> high - low == I.size*I.itemsize
True
>>> I = np.eve(2, dtype='G'); I.dtype
dtype('complex192')
>>> low, high = np.byte_bounds(I)
>>> high - low == I.size*I.itemsize
True
```

**can\_cast(...)**

`can_cast`(from, totype, casting = 'safe')

Returns True if cast between data types can occur according to the casting rule. If from is a scalar or array scalar, also returns True if the scalar value can be cast without overflow or truncation to an `integer`.

## Parameters

from : `dtype`, `dtype` specifier, scalar, or array  
Data type, scalar, or array to cast from.  
totype : `dtype` or `dtype` specifier  
Data type to cast to.  
casting : {'no', 'equiv', 'safe', 'same\_kind', 'unsafe'}, optional  
Controls what kind of data casting may occur.

- \* 'no' means the data types should not be cast at all.
- \* 'equiv' means only `byte`-order changes are allowed.
- \* 'safe' means only casts which can preserve values are allowed.
- \* 'same\_kind' means only safe casts or casts within a kind, like `float64` to `float32`, are allowed.
- \* 'unsafe' means any data conversions may be done.

## Returns

out : bool  
True if cast can occur according to the casting rule.

## Notes

Starting in NumPy 1.9, `can_cast` function now returns False in 'safe' casting mode for `integer/float dtype` and string `dtype` if the string `dtype` length is not long enough to store the max `integer/float` value converted to a string. Previously `can_cast` in 'safe' mode returned True for

[integer/float dtype](#) and a string [dtype](#) of any length.

See also

-----

[dtype](#), [result\\_type](#)

Examples

-----

Basic examples

```
>>> np.can_cast(np.int32, np.int64)
True
>>> np.can_cast(np.float64, np.complex)
True
>>> np.can_cast(np.complex, np.float)
False

>>> np.can_cast('i8', 'f8')
True
>>> np.can_cast('i8', 'f4')
False
>>> np.can_cast('i4', 'S4')
False
```

Casting scalars

```
>>> np.can_cast(100, 'i1')
True
>>> np.can_cast(150, 'i1')
False
>>> np.can_cast(150, 'u1')
True

>>> np.can_cast(3.5e100, np.float32)
False
>>> np.can_cast(1000.0, np.float32)
True
```

Array scalar checks the value, array does not

```
>>> np.can_cast(np.array(1000.0), np.float32)
True
>>> np.can_cast(np.array([1000.0]), np.float32)
False
```

Using the casting rules

```
>>> np.can_cast('i8', 'i8', 'no')
True
>>> np.can_cast('<i8', '>i8', 'no')
False

>>> np.can_cast('<i8', '>i8', 'equiv')
True
>>> np.can_cast('<i4', '>i8', 'equiv')
False

>>> np.can_cast('<i4', '>i8', 'safe')
True
>>> np.can_cast('<i8', '>i4', 'safe')
False

>>> np.can_cast('<i8', '>i4', 'same_kind')
True
>>> np.can_cast('<i8', '>u4', 'same_kind')
False

>>> np.can_cast('<i8', '>u4', 'unsafe')
True
```

**choose(a, choices, out=None, mode='raise')**

Construct an array from an index array and a set of arrays to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = `numpy.lib.index\_tricks`):

```
``np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)])``.
```

But this omits some subtleties. Here is a fully general summary:

Given an "index" array (`a`) of integers and a sequence of `n` arrays (`'choices'`), `a` and each choice array are first [broadcast](#), as necessary, to arrays of a common shape; calling these `*Ba*` and `*Bchoices[i]`, `i = 0,...,n-1` we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each `'i'`. Then, a new array with shape ```Ba.shape``` is created as follows:

```

* if ``mode=raise`` (the default), then, first of all, each element of
 `a` (and thus `Ba`) must be in the range `[0, n-1]`; now, suppose that
 `i` (in that range) is the value at the `(j0, j1, ..., jm)` position
 in `Ba` - then the value at the same position in the new array is the
 value in `Bchoices[i]` at that same position;

* if ``mode=wrap``, values in `a` (and thus `Ba`) may be any (signed)
 integer; modular arithmetic is used to map integers outside the range
 `[0, n-1]` back into that range; and then the new array is constructed
 as above;

* if ``mode=clip``, values in `a` (and thus `Ba`) may be any (signed)
 integer; negative integers are mapped to 0; values greater than `n-1`
 are mapped to `n-1`; and then the new array is constructed as above.

Parameters

a : int array
 This array must contain integers in `[0, n-1]`, where `n` is the number
 of choices, unless ``mode=wrap`` or ``mode=clip``, in which cases any
 integers are permissible.
choices : sequence of arrays
 Choice arrays. `a` and all of the choices must be broadcastable to the
 same shape. If `choices` is itself an array (not recommended), then
 its outermost dimension (i.e., the one corresponding to
    ```choices.shape[0]``) is taken as defining the "sequence".
out : array, optional
    If provided, the result will be inserted into this array. It should
    be of the appropriate shape and dtype.
mode : {'raise' (default), 'wrap', 'clip'}, optional
    Specifies how indices outside `[0, n-1]` will be treated:
        * 'raise' : an exception is raised
        * 'wrap' : value becomes value mod `n`
        * 'clip' : values < 0 are mapped to 0, values > n-1 are mapped to n-1

Returns
-----
merged_array : array
    The merged result.

Raises
-----
ValueError: shape mismatch
    If `a` and each choice array are not all broadcastable to the same
    shape.

See Also
-----
ndarray.choose : equivalent method

Notes
-----
To reduce the chance of misinterpretation, even though the following
"abuse" is nominally supported, `choices` should neither be, nor be
thought of as, a single array, i.e., the outermost sequence-like container
should be either a list or a tuple.

Examples
-----
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... )
array([20, 31, 12, 3])
>>> np.choose([2, 4, 1, 0], choices, mode='clip') # 4 goes to 3 (4-1)
array([20, 31, 12, 3])
>>> # because there are 4 choice arrays
>>> np.choose([2, 4, 1, 0], choices, mode='wrap') # 4 goes to (4 mod 4)
array([20, 1, 12, 3])
>>> # i.e., 0

A couple examples illustrating how choose broadcasts:

>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> np.choose(a, choices)
array([[ 10, -10,  10],
       [-10,  10, -10],
       [ 10, -10,  10]])

>>> # With thanks to Anne Archibald
>>> a = np.array([0, 1]).reshape((2,1,1))
>>> c1 = np.array([1, 2, 3]).reshape((1,3,1))
>>> c2 = np.array([-1, -2, -3, -4, -5]).reshape((1,1,5))

```

```
>>> np.choose(a, (c1, c2)) # result is 2x3x5, res[0,:,:]=c1, res[1,:,:]=c2
array([[[ 1,  1,  1,  1,  1],
       [ 2,  2,  2,  2,  2],
       [ 3,  3,  3,  3,  3]],
      [[-1, -2, -3, -4, -5],
       [-1, -2, -3, -4, -5],
       [-1, -2, -3, -4, -5]])
```

clip(a, a_min, a_max, out=None)

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of ``[0, 1]`` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters

a : array_like

Array containing elements to clip.

a_min : scalar or array_like

Minimum value.

a_max : scalar or array_like

Maximum value. If `a_min` or `a_max` are array_like, then they will be broadcasted to the shape of `a`.

out : ndarray, optional

The results will be placed in this array. It may be the input array for in-place clipping. `out` must be of the right shape to hold the output. Its type is preserved.

Returns

clipped_array : ndarray

An array with the elements of `a`, but where values < `a_min` are replaced with `a_min`, and those > `a_max` with `a_max`.

See Also

numpy.doc.ufuncs : Section "Output arguments"

Examples

```
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3,4,1,1,1,4,4,4,4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

column_stack(tup)

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a **single** 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

Parameters

tup : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

Returns

stacked : 2-D array

The array formed by stacking the given arrays.

See Also

hstack, vstack, concatenate

Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

common_type(*arrays)

Return a scalar type which is common to the input arrays.

The return type will always be an [inexact](#) (i.e. [floating](#) point) scalar type, even if all the arrays are [integer](#) arrays. If one of the inputs is an [integer](#) array, the minimum precision type that is returned is a 64-bit [floating](#) point [dtype](#).

All input arrays can be safely cast to the returned [dtype](#) without loss of information.

Parameters

array1, array2, ... : ndarray
Input arrays.

Returns

out : data type code
Data type code.

See Also

[dtype](#), [mintypecode](#)

Examples

```
>>> np.common_type(np.arange(2, dtype=np.float32))
<type 'numpy.float32'>
>>> np.common_type(np.arange(2, dtype=np.float32), np.arange(2))
<type 'numpy.float64'>
>>> np.common_type(np.arange(4), np.array([45, 6.j]), np.array([45.0]))
<type 'numpy.complex128'>
```

compare_chararrays(...)

compress(condition, a, axis=None, out=None)

Return selected slices of an array along given axis.

When working along a given axis, a slice along that axis is returned in `output` for each index where `condition` evaluates to True. When working on a 1-D array, `compress` is equivalent to `extract`.

Parameters

condition : 1-D array of bools
Array that selects which entries to return. If len(condition) is less than the size of `a` along the given axis, then output is truncated to the length of the condition array.

a : array_like
Array from which to extract a part.

axis : [int](#), optional
Axis along which to take slices. If None (default), work on the flattened array.

out : [ndarray](#), optional
Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns

compressed_array : [ndarray](#)
A copy of `a` without the slices along axis for which `condition` is false.

See Also

[take](#), [choose](#), [diag](#), [diagonal](#), [select](#)
[ndarray.compress](#) : Equivalent method in [ndarray](#)
[np.extract](#) : Equivalent method when working on 1-D arrays
[numpy.doc.ufuncs](#) : Section "Output arguments"

Examples

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.compress([0, 1], a, axis=0)
array([[3, 4]])
>>> np.compress([False, True, True], a, axis=0)
array([[3, 4],
       [5, 6]])
>>> np.compress([False, True], a, axis=1)
array([[2],
       [4],
       [6]])
```

Working on the flattened array does not return slices along an axis but selects elements.

```

>>> np.compress([False, True], a)
array([2])

concatenate(...)
concatenate((a1, a2, ...), axis=0)

Join a sequence of arrays along an existing axis.

Parameters
-----
a1, a2, ... : sequence of array_like
    The arrays must have the same shape, except in the dimension
    corresponding to `axis` (the first, by default).
axis : int, optional
    The axis along which the arrays will be joined. Default is 0.

Returns
-----
res : ndarray
    The concatenated array.

See Also
-----
ma.concatenate : Concatenate function that preserves input masks.
array_split : Split an array into multiple sub-arrays of equal or
    near-equal size.
split : Split array into a list of multiple sub-arrays of equal size.
hsplit : Split array into multiple sub-arrays horizontally (column wise)
vsplit : Split array into multiple sub-arrays vertically (row wise)
dsplit : Split array into multiple sub-arrays along the 3rd axis (depth).
stack : Stack a sequence of arrays along a new axis.
hstack : Stack arrays in sequence horizontally (column wise)
vstack : Stack arrays in sequence vertically (row wise)
dstack : Stack arrays in sequence depth wise (along third dimension)

Notes
-----
When one or more of the arrays to be concatenated is a MaskedArray,
this function will return a MaskedArray object instead of an ndarray,
but the input masks are *not* preserved. In cases where a MaskedArray
is expected as input, use the ma.concatenate function from the masked
array module instead.

Examples
-----
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array([[1, 2, 5],
       [3, 4, 6]])

This function will not preserve masking of MaskedArray inputs.

>>> a = np.ma.arange(3)
>>> a[1] = np.ma.masked
>>> b = np.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
             fill_value = 999999)
>>> b
array([2, 3, 4])
>>> np.concatenate([a, b])
masked_array(data = [0 1 2 2 3 4],
             mask = False,
             fill_value = 999999)
>>> np.ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False],
             fill_value = 999999)

convolve(a, v, mode='full')
Returns the discrete, linear convolution of two one-dimensional sequences.

The convolution operator is often seen in signal processing, where it
models the effect of a linear time-invariant system on a signal [1]_. In
probability theory, the sum of two independent random variables is
distributed according to the convolution of their individual
distributions.

If `v` is longer than `a`, the arrays are swapped before computation.

Parameters

```

```

-----
a : (N,) array_like
    First one-dimensional input array.
v : (M,) array_like
    Second one-dimensional input array.
mode : {'full', 'valid', 'same'}, optional
    'full':
        By default, mode is 'full'. This returns the convolution
        at each point of overlap, with an output shape of (N+M-1,). At
        the end-points of the convolution, the signals do not overlap
        completely, and boundary effects may be seen.

    'same':
        Mode 'same' returns output of length ``max(M, N)``. Boundary
        effects are still visible.

    'valid':
        Mode 'valid' returns output of length
        ``max(M, N) - min(M, N) + 1``. The convolution product is only given
        for points where the signals overlap completely. Values outside
        the signal boundary have no effect.

Returns
-----
out : ndarray
    Discrete, linear convolution of `a` and `v`.

See Also
-----
scipy.signal.fftconvolve : Convolve two arrays using the Fast Fourier
    Transform.
scipy.linalg.toeplitz : Used to construct the convolution operator.
polymul : Polynomial multiplication. Same output as convolve, but also
    accepts poly1d objects as input.

Notes
-----
The discrete convolution operation is defined as

.. math:: (a * v)[n] = \sum_{m=-\infty}^{\infty} a[m] v[n - m]

It can be shown that a convolution :math:`x(t) * y(t)` in time/space
is equivalent to the multiplication :math:`X(f) Y(f)` in the Fourier
domain, after appropriate padding (padding is necessary to prevent
circular convolution). Since multiplication is more efficient (faster)
than convolution, the function `scipy.signal.fftconvolve` exploits the
FFT to calculate the convolution of large data-sets.

References
-----
.. [1] Wikipedia, "Convolution", http://en.wikipedia.org/wiki/Convolution.
```

Examples

Note how the convolution operator flips the second array before "sliding" the two across one another:

```
>>> np.convolve([1, 2, 3], [0, 1, 0.5])
array([ 0.,  1.,  2.5,  4.,  1.5])
```

Only return the middle values of the convolution.
Contains boundary effects, where zeros are taken into account:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'same')
array([ 1.,  2.5,  4. ])
```

The two arrays are of the same length, so there is only one position where they completely overlap:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'valid')
array([ 2.5])
```

copy(a, order='K')
Return an array copy of the given [object](#).

Parameters

a : array_like
Input data.

order : {'C', 'F', 'A', 'K'}, optional
Controls the memory layout of the copy. 'C' means C-order,
'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
'C' otherwise. 'K' means match the layout of `a` as closely
as possible. (Note that this function and :meth:[ndarray](#).copy are very
similar, but have different default values for their order= arguments.)

Returns

```
-----
arr : ndarray
    Array interpretation of `a`.

Notes
-----
This is equivalent to

>>> np.array(a, copy=True) #doctest: +SKIP

Examples
-----
Create an array x, with a reference y and a copy z:

>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)

Note that, when we modify x, y changes, but not z:

>>> x[0] = 10
>>> x[0] == y[0]
True
>>> x[0] == z[0]
False

copyto(...)
copyto(dst, src, casting='same_kind', where=None)

Copies values from one array to another, broadcasting as necessary.

Raises a TypeError if the 'casting' rule is violated, and if
`where` is provided, it selects which elements to copy.

... versionadded:: 1.7.0

Parameters
-----
dst : ndarray
    The array into which values are copied.
src : array_like
    The array from which values are copied.
casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
    Controls what kind of data casting may occur when copying.

    * 'no' means the data types should not be cast at all.
    * 'equiv' means only byte-order changes are allowed.
    * 'safe' means only casts which can preserve values are allowed.
    * 'same_kind' means only safe casts or casts within a kind,
      like float64 to float32, are allowed.
    * 'unsafe' means any data conversions may be done.
where : array_like of bool, optional
    A boolean array which is broadcasted to match the dimensions
    of 'dst', and selects elements to copy from 'src' to 'dst'
    wherever it contains the value True.

corrcoef(x, y=None, rowvar=1, bias=<class numpy._NoValue>, ddof=<class numpy._NoValue>)
Return Pearson product-moment correlation coefficients.

Please refer to the documentation for 'cov' for more detail. The
relationship between the correlation coefficient matrix, 'R', and the
covariance matrix, 'C', is

... math:: R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} C_{jj}}}

The values of 'R' are between -1 and 1, inclusive.

Parameters
-----
x : array_like
    A 1-D or 2-D array containing multiple variables and observations.
    Each row of 'x' represents a variable, and each column a single
    observation of all those variables. Also see 'rowvar' below.
y : array_like, optional
    An additional set of variables and observations. 'y' has the same
    shape as 'x'.
rowvar : int, optional
    If 'rowvar' is non-zero (default), then each row represents a
    variable, with observations in the columns. Otherwise, the relationship
    is transposed: each column represents a variable, while the rows
    contain observations.
bias : _NoValue, optional
    Has no effect, do not use.

... deprecated:: 1.10.0
ddof : _NoValue, optional
    Has no effect, do not use.
```

```

.. deprecated:: 1.10.0

>Returns
-----
R : ndarray
    The correlation coefficient matrix of the variables.

See Also
-----
cov : Covariance matrix

Notes
-----
Due to floating point rounding the resulting array may not be Hermitian,
the diagonal elements may not be 1, and the elements may not satisfy the
inequality  $\text{abs}(a) \leq 1$ . The real and imaginary parts are clipped to the
interval  $[-1, 1]$  in an attempt to improve on that situation but is not
much help in the complex case.

This function accepts but discards arguments `bias` and `ddof`. This is
for backwards compatibility with previous versions of this function. These
arguments had no effect on the return values of the function and can be
safely ignored in this and previous versions of numpy.

correlate(a, v, mode='valid')
Cross-correlation of two 1-dimensional sequences.

This function computes the correlation as generally defined in signal
processing texts::

     $c_{\{av\}}[k] = \sum_n a[n+k] * \text{conj}(v[n])$ 

with  $a$  and  $v$  sequences being zero-padded where necessary and  $\text{conj}$  being
the conjugate.

Parameters
-----
a, v : array_like
    Input sequences.
mode : {'valid', 'same', 'full'}, optional
    Refer to the `convolve` docstring. Note that the default
    is 'valid', unlike `convolve`, which uses 'full'.
old_behavior : bool
    'old_behavior' was removed in NumPy 1.10. If you need the old
    behavior, use `multiarray.correlate`.

>Returns
-----
out : ndarray
    Discrete cross-correlation of `a` and `v`.

See Also
-----
convolve : Discrete, linear convolution of two one-dimensional sequences.
multiarray.correlate : Old, no conjugate, version of correlate.

Notes
-----
The definition of correlation above is not unique and sometimes correlation
may be defined differently. Another common definition is::

     $c'_{\{av\}}[k] = \sum_n a[n] \text{conj}(v[n+k])$ 

which is related to `` $c_{\{av\}}[k]$ `` by `` $c'_{\{av\}}[k] = c_{\{av\}}[-k]$ ``.

Examples
-----
>>> np.correlate([1, 2, 3], [0, 1, 0.5])
array([ 3.5])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "same")
array([ 2., 3.5, 3. ])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "full")
array([ 0.5, 2., 3.5, 3., 0. ])

Using complex sequences:

>>> np.correlate([1+1j, 2, 3-1j], [0, 1, 0.5j], 'full')
array([ 0.5-0.5j, 1.0+0.j, 1.5-1.5j, 3.0-1.j, 0.0+0.j ])

Note that you get the time reversed, complex conjugated result
when the two input sequences change places, i.e.,
`` $c_{\{va\}}[k] = c^*_{\{av\}}[-k]$ ``

>>> np.correlate([0, 1, 0.5j], [1+1j, 2, 3-1j], 'full')
array([ 0.0+0.j, 3.0+1.j, 1.5+1.5j, 1.0+0.j, 0.5+0.5j])

count_nonzero(...)
count nonzero(a)

```

```

Counts the number of non-zero values in the array ``a``.

Parameters
-----
a : array_like
    The array for which to count non-zeros.

Returns
-----
count : int or array of int
    Number of non-zero values in the array.

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

Examples
-----
>>> np.count nonzero(np.eye(4))
4
>>> np.count nonzero([[0,1,7,0,0],[3,0,0,2,19]])
5

cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None, aweights=None)
Estimate a covariance matrix, given data and weights.

Covariance indicates the level to which two variables vary together.
If we examine N-dimensional samples, :math:`X = [x_1, x_2, \dots x_N]^T` ,
then the covariance matrix element :math:`C_{ij}` is the covariance of
:math:`x_i` and :math:`x_j` . The element :math:`C_{ii}` is the variance
of :math:`x_i` .

See the notes for an outline of the algorithm.

Parameters
-----
m : array_like
    A 1-D or 2-D array containing multiple variables and observations.
    Each row of `m` represents a variable, and each column a single
observation of all those variables. Also see `rowvar` below.
y : array_like, optional
    An additional set of variables and observations. `y` has the same form
    as that of `m` .
rowvar : bool, optional
    If `rowvar` is True (default), then each row represents a
    variable, with observations in the columns. Otherwise, the relationship
    is transposed: each column represents a variable, while the rows
    contain observations.
bias : bool, optional
    Default normalization (False) is by ``N - 1``, where ``N`` is the
    number of observations given (unbiased estimate). If `bias` is True, then
    normalization is by ``N`` . These values can be overridden by using the
    keyword ``ddof`` in numpy versions >= 1.5.
ddof : int, optional
    If not ``None`` the default value implied by `bias` is overridden.
    Note that ``ddof=1`` will return the unbiased estimate, even if both
    `fweights` and `aweights` are specified, and ``ddof=0`` will return
    the simple average. See the notes for the details. The default value
    is ``None`` .
.. versionadded:: 1.5
fweights : array_like, int, optional
    1-D array of integer frequency weights; the number of times each
    observation vector should be repeated.

.. versionadded:: 1.10
aweights : array_like, optional
    1-D array of observation vector weights. These relative weights are
    typically large for observations considered "important" and smaller for
    observations considered less "important". If ``ddof=0`` the array of
    weights can be used to assign probabilities to observation vectors.

.. versionadded:: 1.10

Returns
-----
out : ndarray
    The covariance matrix of the variables.

See Also
-----
corrcoef : Normalized covariance matrix

Notes
-----
Assume that the observations are in the columns of the observation
array `m` and let ``f = fweights`` and ``a = aweights`` for brevity. The
steps to compute the weighted covariance are as follows::
```

```

>>> w = f * a
>>> v1 = np.sum(w)
>>> v2 = np.sum(w * a)
>>> m -= np.sum(m * w, axis=1, keepdims=True) / v1
>>> cov = np.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)

Note that when ``a == 1``, the normalization factor
``v1 / (v1**2 - ddof * v2)`` goes over to ``1 / (np.sum(f) - ddof)``
as it should.

Examples
-----
Consider two variables, :math:`x_0` and :math:`x_1`, which
correlate perfectly, but in opposite directions:

>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])

Note how :math:`x_0` increases while :math:`x_1` decreases. The covariance
matrix shows this clearly:

>>> np.cov(x)
array([[ 1., -1.],
       [-1.,  1.]])

Note that element :math:`C_{0,1}` , which shows the correlation between
:math:`x_0` and :math:`x_1` , is negative.

Further, note how `x` and `y` are combined:

>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = np.vstack((x,y))
>>> print(np.cov(X))
[[ 11.71      -4.286      ]
 [ -4.286      2.14413333]]
>>> print(np.cov(x, y))
[[ 11.71      -4.286      ]
 [ -4.286      2.14413333]]
>>> print(np.cov(x))
11.71

cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)
Return the cross product of two (arrays of) vectors.

The cross product of `a` and `b` in :math:`R^3` is a vector perpendicular
to both `a` and `b`. If `a` and `b` are arrays of vectors, the vectors
are defined by the last axis of `a` and `b` by default, and these axes
can have dimensions 2 or 3. Where the dimension of either `a` or `b` is
2, the third component of the input vector is assumed to be zero and the
cross product calculated accordingly. In cases where both input vectors
have dimension 2, the z-component of the cross product is returned.

Parameters
-----
a : array_like
    Components of the first vector(s).
b : array_like
    Components of the second vector(s).
axisa : int, optional
    Axis of `a` that defines the vector(s). By default, the last axis.
axisb : int, optional
    Axis of `b` that defines the vector(s). By default, the last axis.
axisc : int, optional
    Axis of `c` containing the cross product vector(s). Ignored if
    both input vectors have dimension 2, as the return is scalar.
    By default, the last axis.
axis : int, optional
    If defined, the axis of `a`, `b` and `c` that defines the vector(s)
    and cross product(s). Overrides `axisa`, `axisb` and `axisc`.

Returns
-----
c : ndarray
    Vector cross product(s).

Raises
-----
ValueError
    When the dimension of the vector(s) in `a` and/or `b` does not
    equal 2 or 3.

See Also
-----
inner : Inner product
outer : Outer product.

```

```

ix_ : Construct index arrays.

Notes
-----
.. versionadded:: 1.9.0

Supports full broadcasting of the inputs.

Examples
-----
Vector cross-product.

>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([-3, 6, -3])

One vector with dimension 2.

>>> x = [1, 2]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([12, -6, -3])

Equivalently:

>>> x = [1, 2, 0]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([12, -6, -3])

Both vectors with dimension 2.

>>> x = [1,2]
>>> y = [4,5]
>>> np.cross(x, y)
-3

Multiple vector cross-products. Note that the direction of the cross
product vector is defined by the 'right-hand rule'.

>>> x = np.array([[1,2,3], [4,5,6]])
>>> y = np.array([[4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[ -3, 6, -3],
       [ 3, -6, 3]])

The orientation of 'c' can be changed using the 'axisc' keyword.

>>> np.cross(x, y, axisc=0)
array([[ -3, 3],
       [ 6, -6],
       [-3, 3]])

Change the vector definition of 'x' and 'y' using 'axisa' and 'axisb'.

>>> x = np.array([[1,2,3], [4,5,6], [7, 8, 9]])
>>> y = np.array([[7, 8, 9], [4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[ -6, 12, -6],
       [ 0, 0, 0],
       [ 6, -12, 6]])
>>> np.cross(x, y, axisa=0, axisb=0)
array([[-24, 48, -24],
       [-30, 60, -30],
       [-36, 72, -36]])

cumprod(a, axis=None, dtype=None, out=None)
Return the cumulative product of elements along a given axis.

Parameters
-----
a : array_like
    Input array.
axis : int, optional
    Axis along which the cumulative product is computed. By default
    the input is flattened.
dtype : dtype, optional
    Type of the returned array, as well as of the accumulator in which
    the elements are multiplied. If *dtype* is not specified, it
    defaults to the dtype of `a`, unless `a` has an integer dtype with
    a precision less than that of the default platform integer. In
    that case, the default platform integer is used instead.
out : ndarray, optional
    Alternative output array in which to place the result. It must
    have the same shape and buffer length as the expected output
    but the type of the resulting values will be cast if necessary.

Returns
-----
```

```
-----  
cumprod : ndarray  
    A new array holding the result is returned unless `out` is  
    specified, in which case a reference to out is returned.  
  
See Also  
-----  
numpy.doc.ufuncs : Section "Output arguments"  
  
Notes  
-----  
Arithmetic is modular when using integer types, and no error is  
raised on overflow.  
  
Examples  
-----  
>>> a = np.array([1,2,3])  
>>> np.cumprod(a) # intermediate results 1, 1*2  
...  
# total product 1*2*3 = 6  
array([1, 2, 6])  
>>> a = np.array([[1, 2, 3], [4, 5, 6]])  
>>> np.cumprod(a, dtype=float) # specify type of output  
array([[ 1.,   2.,   6.,  24., 120., 720.]])  
  
The cumulative product for each column (i.e., over the rows) of `a`:  
  
>>> np.cumprod(a, axis=0)  
array([[ 1,  2,  3],  
       [ 4, 10, 18]])  
  
The cumulative product for each row (i.e. over the columns) of `a`:  
  
>>> np.cumprod(a, axis=1)  
array([[ 1,  2,  6],  
       [ 4, 20, 120]])  
  
cumproduct(a, axis=None, dtype=None, out=None)  
Return the cumulative product over the given axis.  
  
See Also  
-----  
cumprod : equivalent function; see for details.  
  
cumsum(a, axis=None, dtype=None, out=None)  
Return the cumulative sum of the elements along a given axis.  
  
Parameters  
-----  
a : array_like  
    Input array.  
axis : int, optional  
    Axis along which the cumulative sum is computed. The default  
(None) is to compute the cumsum over the flattened array.  
dtype : dtype, optional  
    Type of the returned array and of the accumulator in which the  
elements are summed. If dtype is not specified, it defaults  
to the dtype of `a`, unless `a` has an integer dtype with a  
precision less than that of the default platform integer. In  
that case, the default platform integer is used.  
out : ndarray, optional  
    Alternative output array in which to place the result. It must  
have the same shape and buffer length as the expected output  
but the type will be cast if necessary. See `doc.ufuncs`  
(Section "Output arguments") for more details.  
  
Returns  
-----  
cumsum_along_axis : ndarray  
    A new array holding the result is returned unless `out` is  
specified, in which case a reference to `out` is returned. The  
result has the same size as `a`, and the same shape as `a` if  
`axis` is not None or `a` is a 1-d array.  
  
See Also  
-----  
sum : Sum array elements.  

```

Examples

```
-----
>>> a = np.array([[1,2,3], [4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.cumsum(a)
array([ 1,  3,  6, 10, 15, 21])
>>> np.cumsum(a, dtype=float)      # specifies type of output value(s)
array([ 1.,   3.,   6.,  10.,  15.,  21.])

>>> np.cumsum(a, axis=0)         # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a, axis=1)         # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

datetime_as_string(...)

datetime_data(...)

delete(arr, obj, axis=None)

Return a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by `arr[obj]`.

Parameters

```
-----
arr : array_like
    Input array.
obj : slice, int or array of ints
    Indicate which sub-arrays to remove.
axis : int, optional
    The axis along which to delete the subarray defined by `obj`.
    If `axis` is None, `obj` is applied to the flattened array.
```

Returns

```
-----
out : ndarray
    A copy of `arr` with the elements specified by `obj` removed. Note that `delete` does not occur in-place. If `axis` is None, `out` is a flattened array.
```

See Also

```
-----
insert : Insert elements into an array.
append : Append elements at the end of an array.
```

Notes

Often it is preferable to use a boolean mask. For example:

```
>>> mask = np.ones(len(arr), dtype=bool)
>>> mask[[0,2,4]] = False
>>> result = arr[mask,...]
```

Is equivalent to `np.delete(arr, [0,2,4], axis=0)`, but allows further use of `mask`.

Examples

```
-----
>>> arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.delete(arr, 1, 0)
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])

>>> np.delete(arr, np.s_[:2], 1)
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>> np.delete(arr, [1,3,5], None)
array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
```

deprecate(*args, **kwargs)

Issues a [DeprecationWarning](#), adds warning to `old_name`'s docstring, rebinds ``old_name.__name__`` and returns the new function [object](#).

This function may also be used as a decorator.

Parameters

```
-----
func : function
```

```
The function to be deprecated.  
old_name : str, optional  
    The name of the function to be deprecated. Default is None, in  
    which case the name of `func` is used.  
new_name : str, optional  
    The new name for the function. Default is None, in which case the  
    deprecation message is that 'old_name' is deprecated. If given, the  
    deprecation message is that 'old_name' is deprecated and 'new_name'  
    should be used instead.  
message : str, optional  
    Additional explanation of the deprecation. Displayed in the  
    docstring after the warning.  
  
Returns  
-----  
old_func : function  
    The deprecated function.  
  
Examples  
-----  
Note that ``olduint`` returns a value after printing Deprecation  
Warning:  
  
>>> olduint = np.deprecate(np.uint)  
>>> olduint(6)  
/usr/lib/python2.5/site-packages/numpy/lib/utils.py:114:  
DeprecationWarning: uint32 is deprecated  
    warnings.warn(str1, DeprecationWarning)  
6  
  
deprecate_with_doc lambda msg  
  
diag(v, k=0)  
Extract a diagonal or construct a diagonal array.  
  
See the more detailed documentation for ``numpy.diagonal`` if you use this  
function to extract a diagonal and wish to write to the resulting array;  
whether it returns a copy or a view depends on what version of numpy you  
are using.  
  
Parameters  
-----  
v : array_like  
    If 'v' is a 2-D array, return a copy of its 'k'-th diagonal.  
    If 'v' is a 1-D array, return a 2-D array with 'v' on the 'k'-th  
    diagonal.  
k : int, optional  
    Diagonal in question. The default is 0. Use 'k>0' for diagonals  
    above the main diagonal, and 'k<0' for diagonals below the main  
    diagonal.  
  
Returns  
-----  
out : ndarray  
    The extracted diagonal or constructed diagonal array.  
  
See Also  
-----  
diagonal : Return specified diagonals.  
diagflat : Create a 2-D array with the flattened input as a diagonal.  
trace : Sum along diagonals.  
triu : Upper triangle of an array.  
tril : Lower triangle of an array.  
  
Examples  
-----  
>>> x = np.arange(9).reshape((3,3))  
>>> x  
array([[0, 1, 2],  
         [3, 4, 5],  
         [6, 7, 8]])  
  
>>> np.diag(x)  
array([0, 4, 8])  
>>> np.diag(x, k=1)  
array([1, 5])  
>>> np.diag(x, k=-1)  
array([3, 7])  
  
>>> np.diag(np.diag(x))  
array([[0, 0, 0],  
         [0, 4, 0],  
         [0, 0, 8]])  
  
diag_indices(n, ndim=2)  
Return the indices to access the main diagonal of an array.  
  
This returns a tuple of indices that can be used to access the main
```

```

diagonal of an array `a` with ``a.ndim >= 2`` dimensions and shape
(n, n, ..., n). For ``a.ndim = 2`` this is the usual diagonal, for
``a.ndim > 2`` this is the set of indices to access ``a[i, i, ..., i]``
for ``i = [0..n-1]``.

Parameters
-----
n : int
    The size, along each dimension, of the arrays for which the returned
    indices can be used.

ndim : int, optional
    The number of dimensions.

See also
-----
diag\_indices\_from

Notes
-----
.. versionadded:: 1.4.0

Examples
-----
Create a set of indices to access the diagonal of a (4, 4) array:

>>> di = np.diag\_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11],
           [12, 13, 14, 15]])
>>> a[di] = 100
>>> a
array([[100,  1,  2,  3],
           [ 4, 100,  6,  7],
           [ 8,  9, 100, 11],
           [12, 13, 14, 100]])

Now, we create indices to manipulate a 3-D array:

>>> d3 = np.diag\_indices(2, 3)
>>> d3
(array([0, 1]), array([0, 1]), array([0, 1]))

And use it to set the diagonal of an array of zeros to 1:

>>> a = np.zeros((2, 2, 2), dtype=np.int)
>>> a[d3] = 1
>>> a
array([[[1, 0],
           [0, 0]],
          [[0, 0],
           [0, 1]]])

diag\_indices\_from(arr)
Return the indices to access the main diagonal of an n-dimensional array.

See 'diag_indices' for full details.

Parameters
-----
arr : array, at least 2-D

See Also
-----
diag\_indices

Notes
-----
.. versionadded:: 1.4.0

diagflat(v, k=0)
Create a two-dimensional array with the flattened input as a diagonal.

Parameters
-----
v : array_like
    Input data, which is flattened and set as the `k`-th
    diagonal of the output.
k : int, optional
    Diagonal to set; 0, the default, corresponds to the "main" diagonal,
    a positive (negative) `k` giving the number of the diagonal above
    (below) the main.
```

Returns

`out : ndarray`
The 2-D output array.

See Also

`diag` : MATLAB work-alike for 1-D and 2-D arrays.
`diagonal` : Return specified diagonals.
`trace` : Sum along diagonals.

Examples

`>>> np.diagflat([[1,2], [3,4]])`
`array([[1, 0, 0, 0],`
 `[0, 2, 0, 0],`
 `[0, 0, 3, 0],`
 `[0, 0, 0, 4]])`

`>>> np.diagflat([1,2], 1)`
`array([[0, 1, 0],`
 `[0, 0, 2],`
 `[0, 0, 0]])`

diagonal(a, offset=0, axis1=0, axis2=1)
Return specified diagonals.

If `'a'` is 2-D, returns the diagonal of `'a'` with the given offset, i.e., the collection of elements of the form `'a[i, i+offset]'`. If `'a'` has more than two dimensions, then the axes specified by `'axis1'` and `'axis2'` are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing `'axis1'` and `'axis2'` and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning is issued.

Starting in NumPy 1.9 it returns a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In some future release, it will return a read/write view and writing to the returned array will alter your original array. The returned array will have the same type as the input array.

If you don't write to the array returned by this function, then you can just ignore all of the above.

If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `'np.diagonal(a).copy()'` instead of just `'np.diagonal(a)'`. This will work with both past and future versions of NumPy.

Parameters

`a : array_like`
 Array from which the diagonals are taken.
`offset : int, optional`
 Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).
`axis1 : int, optional`
 Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).
`axis2 : int, optional`
 Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).

Returns

`array_of_diagonals : ndarray`
 If `'a'` is 2-D and not a `matrix`, a 1-D array of the same type as `'a'` containing the diagonal is returned. If `'a'` is a `matrix`, a 1-D array containing the diagonal is returned in order to maintain backward compatibility. If the dimension of `'a'` is greater than two, then an array of diagonals is returned, "packed" from left-most dimension to right-most (e.g., if `'a'` is 3-D, then the diagonals are "packed" along rows).

Raises

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

See Also

```

diag : MATLAB work-a-like for 1-D and 2-D arrays.
diagflat : Create diagonal arrays.
trace : Sum along diagonals.

Examples
-----
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])

A 3-D example:

>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0, # Main diagonals of two arrays created by skipping
...             0, # across the outer(left)-most axis last and
...             1) # the "middle" (row) axis first.
array([[0, 6],
       [1, 7]])

The sub-arrays whose main diagonals we just obtained; note that each
corresponds to fixing the right-most (column) axis, and that the
diagonals are "packed" in rows.

>>> a[:, :, 0] # main diagonal is [0 6]
array([[0, 2],
       [4, 6]])
>>> a[:, :, 1] # main diagonal is [1 7]
array([[1, 3],
       [5, 7]])

diff(a, n=1, axis=-1)
Calculate the n-th discrete difference along given axis.

The first difference is given by ``out[n] = a[n+1] - a[n]`` along
the given axis, higher differences are calculated by using `diff`
recursively.

Parameters
-----
a : array_like
    Input array
n : int, optional
    The number of times values are differenced.
axis : int, optional
    The axis along which the difference is taken, default is the last axis.

Returns
-----
diff : ndarray
    The n-th differences. The shape of the output is the same as `a`
    except along `axis` where the dimension is smaller by `n`.

See Also
-----
gradient, ediff1d, cumsum

Examples
-----
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.diff(x)
array([ 1,  2,  3, -7])
>>> np.diff(x, n=2)
array([-1,  1, -10])

>>> x = np.array([[1, 3, 6, 10], [0, 5, 6, 8]])
>>> np.diff(x)
array([[2, 3, 4],
       [5, 1, 2]])
>>> np.diff(x, axis=0)
array([[-1,  2,  0, -2]])

digitize(...)
digitize(x, bins, right=False)

Return the indices of the bins to which each value in input array belongs.

Each index ``i`` returned is such that ``bins[i-1] <= x < bins[i]`` if
`bins` is monotonically increasing, or ``bins[i-1] > x >= bins[i]`` if

```

```
'bins' is monotonically decreasing. If values in `x` are beyond the
bounds of `bins`, 0 or ``len(bins)`` is returned as appropriate. If right
is True, then the right bin is closed so that the index ``i`` is such
that ``bins[i-1] < x <= bins[i]`` or ``bins[i-1] >= x > bins[i]`` if `bins`
is monotonically increasing or decreasing, respectively.

Parameters
-----
x : array_like
    Input array to be binned. Prior to Numpy 1.10.0, this array had to
    be 1-dimensional, but can now have any shape.
bins : array_like
    Array of bins. It has to be 1-dimensional and monotonic.
right : bool, optional
    Indicating whether the intervals include the right or the left bin
    edge. Default behavior is (right==False) indicating that the interval
    does not include the right edge. The left bin end is open in this
    case, i.e., ``bins[i-1] <= x < bins[i]`` is the default behavior for
    monotonically increasing bins.

Returns
-----
out : ndarray of ints
    Output array of indices, of same shape as `x`.

Raises
-----
ValueError
    If `bins` is not monotonic.
TypeError
    If the type of the input is complex.
See Also
-----
bincount, histogram, unique

Notes
-----
If values in `x` are such that they fall outside the bin range,
attempting to index `bins` with the indices that `digitize` returns
will result in an IndexError.

.. versionadded:: 1.10.0

`np.digitize` is implemented in terms of `np.searchsorted`. This means
that a binary search is used to bin the values, which scales much better
for larger number of bins than the previous linear search. It also removes
the requirement for the input array to be 1-dimensional.

Examples
-----
>>> x = np.array([0.2, 6.4, 3.0, 1.6])
>>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = np.digitize(x, bins)
>>> inds
array([1, 4, 3, 2])
>>> for n in range(x.size):
...     print(bins[inds[n]-1], "<=", x[n], "<", bins[inds[n]])
...
0.0 <= 0.2 < 1.0
4.0 <= 6.4 < 10.0
2.5 <= 3.0 < 4.0
1.0 <= 1.6 < 2.5

>>> x = np.array([1.2, 10.0, 12.4, 15.5, 20.])
>>> bins = np.array([0, 5, 10, 15, 20])
>>> np.digitize(x,bins,right=True)
array([1, 2, 3, 4, 4])
>>> np.digitize(x,bins,right=False)
array([1, 3, 3, 4, 5])

disp(mesg, device=None, linefeed=True)
Display a message on a device.

Parameters
-----
mesg : str
    Message to display.
device : object
    Device to write message. If None, defaults to ``sys.stdout`` which is
    very similar to ``print``. `device` needs to have ``write()`` and
    ``flush()`` methods.
linefeed : bool, optional
    Option whether to print a line feed or not. Defaults to True.

Raises
-----
AttributeError
    If `device` does not have a ``write()`` or ``flush()`` method.
```

Examples

```
-----
Besides ``sys.stdout``, a file-like object can also be used as it has both required methods:
```

```
>>> from StringIO import StringIO
>>> buf = StringIO()
>>> np.disp('Display' in a file', device=buf)
>>> buf.getvalue()
'Display' in a file\n'
```

dot(...)

```
dot(a, b, out=None)
```

Dot product of two arrays.

For 2-D arrays it is equivalent to [matrix](#) multiplication, and for 1-D arrays to inner product of vectors (without [complex](#) conjugation). For N dimensions it is a sum product over the last axis of `a` and the second-to-last of `b`:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:]* b[k,:,:m])
```

Parameters

```
-----
a : array_like
    First argument.
b : array_like
    Second argument.
out : ndarray, optional
    Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for 'dot'(a,b). This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.
```

Returns

```
-----
output : ndarray
    Returns the dot product of `a` and `b`. If `a` and `b` are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned.
    If `out` is given, then it is returned.
```

Raises

```
-----
ValueError
    If the last dimension of `a` is not the same size as the second-to-last dimension of `b`.
```

See Also

```
-----
vdot : Complex-conjugating dot product.
tensordot : Sum products over arbitrary axes.
einsum : Einstein summation convention.
matmul : '@' operator as method with out parameter.
```

Examples

```
-----
>>> np.dot(3, 4)
12
```

Neither argument is [complex](#)-conjugated:

```
>>> np.dot([[2j, 3j], [2j, 3j]])
(-13+0j)
```

For 2-D arrays it is the [matrix](#) product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:]* b[1,2,:,:2])
499128
```

dsplit(ary, indices_or_sections)

Split array into multiple sub-arrays along the 3rd axis (depth).

Please refer to the `split` documentation. `dsplit` is equivalent

to `split` with ``axis=2``, the array is always split along the third axis provided the array dimension is greater than or equal to 3.

See Also

split : Split an array into multiple sub-arrays of equal size.

Examples

```
-----
>>> x = np.arange(16.0).reshape(2, 2, 4)
>>> x
array([[[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]],
      [[ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]]])
>>> np.dsplit(x, 2)
[array([[ 0.,  1.],
       [ 4.,  5.]],
      [[ 8.,  9.],
       [12., 13.]]]),
 array([[ 2.,  3.],
       [ 6.,  7.]],
      [[10., 11.],
       [14., 15.]]])
>>> np.dsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
       [ 4.,  5.,  6.]],
      [[ 8.,  9., 10.],
       [12., 13., 14.]]),
 array([[ 3.],
       [ 7.]],
      [[11.],
       [15.]]]),
 array([], dtype=float64)]
```

dstack(tup)

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a [single](#) array. Rebuilds arrays divided by `dsplit`. This is a simple way to stack 2D arrays (images) into a [single](#) 3D array for processing.

Parameters

```
-----
tup : sequence of arrays
    Arrays to stack. All of them must have the same shape along all
    but the third axis.
```

Returns

```
-----
stacked : ndarray
    The array formed by stacking the given arrays.
```

See Also

```
-----
stack : Join a sequence of arrays along a new axis.
vstack : Stack along first axis.
hstack : Stack along second axis.
concatenate : Join a sequence of arrays along an existing axis.
dsplit : Split array along third axis.
```

Notes

Equivalent to ``np.concatenate(tup, axis=2)``.

Examples

```
-----
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

>>> a = np.array(([1],[2],[3]))
>>> b = np.array(([2],[3],[4]))
>>> np.dstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

ediff1d(ary, to_end=None, to_begin=None)

The differences between consecutive elements of an array.

Parameters

```
-----
ary : array_like
```

```
If necessary, will be flattened before the differences are taken.
to_end : array_like, optional
    Number(s) to append at the end of the returned differences.
to_begin : array_like, optional
    Number(s) to prepend at the beginning of the returned differences.

Returns
-----
ediff1d : ndarray
    The differences. Loosely, this is ``ary.flat[1:] - ary.flat[:-1]``.

See Also
-----
diff, gradient

Notes
-----
When applied to masked arrays, this function drops the mask information
if the `to_begin` and/or `to_end` parameters are used.

Examples
-----
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.ediff1d(x)
array([ 1,  2,  3, -7])

>>> np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
array([-99,   1,   2,   3,  -7,  88,  99])

The returned array is always 1D.

>>> y = [[1, 2, 4], [1, 6, 24]]
>>> np.ediff1d(y)
array([ 1,  2, -3,  5, 18])

einsum(...)
einsum(subscripts, *operands, out=None, dtype=None, order='K', casting='safe')

Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional
array operations can be represented in a simple fashion. This function
provides a way to compute such summations. The best way to understand this
function is to try the examples below, which show how many common NumPy
functions can be implemented as calls to `einsum`.

Parameters
-----
subscripts : str
    Specifies the subscripts for summation.
operands : list of array_like
    These are the arrays for the operation.
out : ndarray, optional
    If provided, the calculation is done into this array.
dtype : data-type, optional
    If provided, forces the calculation to use the data type specified.
    Note that you may have to also give a more liberal 'casting'
    parameter to allow the conversions.
order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout of the output. 'C' means it should
    be C contiguous. 'F' means it should be Fortran contiguous,
    'A' means it should be 'F' if the inputs are all 'F', 'C' otherwise.
    'K' means it should be as close to the layout as the inputs as
    is possible, including arbitrarily permuted axes.
    Default is 'K'.
casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
    Controls what kind of data casting may occur. Setting this to
    'unsafe' is not recommended, as it can adversely affect accumulations.

    * 'no' means the data types should not be cast at all.
    * 'equiv' means only byte-order changes are allowed.
    * 'safe' means only casts which can preserve values are allowed.
    * 'same_kind' means only safe casts or casts within a kind,
      like float64 to float32, are allowed.
    * 'unsafe' means any data conversions may be done.

Returns
-----
output : ndarray
    The calculation based on the Einstein summation convention.

See Also
-----
dot, inner, outer, tensordot

Notes
-----
.. versionadded:: 1.6.0
```

The subscripts string is a comma-separated list of subscript labels, where each label refers to a dimension of the corresponding operand. Repeated subscripts labels in one operand take the diagonal. For example, ``np.einsum('ii', a)`` is equivalent to ``np.trace(a)``.

Whenever a label is repeated, it is summed, so ``np.einsum('i,i', a, b)`` is equivalent to ``np.inner(a,b)``. If a label appears only once, it is not summed, so ``np.einsum('i', a)`` produces a view of ``a`` with no changes.

The order of labels in the output is by default alphabetical. This means that ``np.einsum('ij', a)`` doesn't affect a 2D array, while ``np.einsum('ji', a)`` takes its transpose.

The output can be controlled by specifying output subscript labels as well. This specifies the label order, and allows summing to be disallowed or forced when desired. The call ``np.einsum('i->', a)`` is like ``np.sum(a, axis=-1)``, and ``np.einsum('ii->i', a)`` is like ``np.diag(a)``. The difference is that `einsum` does not allow broadcasting by default.

To enable and control broadcasting, use an ellipsis. Default NumPy-style broadcasting is done by adding an ellipsis to the left of each term, like ``np.einsum('...ii->...i', a)``. To take the trace along the first and last axes, you can do ``np.einsum('i...i', a)``, or to do a [matrix-matrix](#) product with the left-most indices instead of rightmost, you can do ``np.einsum('ij...jk...>ik...', a, b)``.

When there is only one operand, no axes are summed, and no output parameter is provided, a view into the operand is returned instead of a new array. Thus, taking the diagonal as ``np.einsum('ii->i', a)`` produces a view.

An alternative way to provide the subscripts and operands is as ``einsum(op0, sublist0, opl, sublist1, ..., [sublistout])``. The examples below have corresponding `einsum` calls with the two parameter methods.

.. versionadded:: 1.10.0

Views returned from einsum are now writeable whenever the input array is writeable. For example, ``np.einsum('ijk...>kji...', a)`` will now have the same effect as ``np.swapaxes(a, 0, 2)`` and ``np.einsum('ii->i', a)`` will return a writeable view of the diagonal of a 2D array.

Examples

```
-----
>>> a = np.arange(25).reshape(5,5)
>>> b = np.arange(5)
>>> c = np.arange(6).reshape(2,3)

>>> np.einsum('ii', a)
60
>>> np.einsum(a, [0,0])
60
>>> np.trace(a)
60

>>> np.einsum('ii->i', a)
array([ 0,  6, 12, 18, 24])
>>> np.einsum(a, [0,0], [0])
array([ 0,  6, 12, 18, 24])
>>> np.diag(a)
array([ 0,  6, 12, 18, 24])

>>> np.einsum('ij,j', a, b)
array([ 30,  80, 130, 180, 230])
>>> np.einsum(a, [0,1], b, [1])
array([ 30,  80, 130, 180, 230])
>>> np.dot(a, b)
array([ 30,  80, 130, 180, 230])
>>> np.einsum('...j,j', a, b)
array([ 30,  80, 130, 180, 230])

>>> np.einsum('ji', c)
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> np.einsum(c, [1,0])
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> c.T
array([[0, 3],
       [1, 4],
       [2, 5]])

>>> np.einsum('..., ...', 3, c)
array([[ 0,   3,   6],
```

```

        [ 9, 12, 15]])
>>> np.einsum(3, [Ellipsis], c, [Ellipsis])
array([[ 0,  3,  6],
       [ 9, 12, 15]])
>>> np.multiply(3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])

>>> np.einsum('i,i', b, b)
30
>>> np.einsum(b, [0], b, [0])
30
>>> np.inner(b,b)
30

>>> np.einsum('i,j', np.arange(2)+1, b)
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> np.einsum(np.arange(2)+1, [0], b, [1])
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> np.outer(np.arange(2)+1, b)
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])

>>> np.einsum('i...->...', a)
array([50, 55, 60, 65, 70])
>>> np.einsum(a, [0, Ellipsis], [Ellipsis])
array([50, 55, 60, 65, 70])
>>> np.sum(a, axis=0)
array([50, 55, 60, 65, 70])

>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> np.einsum('ijk,jil->kli', a, b)
array([[ 4400.,  4730.],
       [ 4532.,  4874.],
       [ 4664.,  5018.],
       [ 4796.,  5162.],
       [ 4928.,  5306.]])
>>> np.einsum(a, [0,1,2], b, [1,0,3], [2,3])
array([[ 4400.,  4730.],
       [ 4532.,  4874.],
       [ 4664.,  5018.],
       [ 4796.,  5162.],
       [ 4928.,  5306.]])
>>> np.tensordot(a,b, axes=[[1,0],[0,1]))
array([[ 4400.,  4730.],
       [ 4532.,  4874.],
       [ 4664.,  5018.],
       [ 4796.,  5162.],
       [ 4928.,  5306.]])

>>> a = np.arange(6).reshape((3,2))
>>> b = np.arange(12).reshape((4,3))
>>> np.einsum('ki,jk->ij', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])
>>> np.einsum('ki,...k->i...', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])
>>> np.einsum('k...,jk', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])

>>> # since version 1.10.0
>>> a = np.zeros((3, 3))
>>> np.einsum('ii->i', a)[:] = 1
>>> a
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

empty(...)
`empty(shape, dtype=float, order='C')`

Return a new array of given shape and type, without initializing entries.

Parameters

`shape : int or tuple of int`
 Shape of the empty array

`dtype : data-type, optional`
 Desired output data-type.

`order : {'C', 'F'}, optional`
 Whether to store multi-dimensional data in row-major
 (C-style) or column-major (Fortran-style) order in
 memory.

Returns

`out : ndarray`
 Array of uninitialized (arbitrary) data of the given shape, `dtype`, and
 order. Object arrays will be initialized to None.

See Also

`empty_like`, `zeros`, `ones`

Notes

`empty`, unlike `zeros`, does not set the array values to zero,
and may therefore be marginally faster. On the other hand, it requires
the user to manually set all the values in the array, and should be
used with caution.

Examples

`>>> np.empty([2, 2])`
`array([[-9.74499359e+001, 6.69583040e-309],`
`[2.13182611e-314, 3.06959433e-309]])` #random

`>>> np.empty([2, 2], dtype=int)`
`array([[-1073741821, -1067949133],`
`[496041986, 19249760]])` #random

`empty_like(...)`
`empty_like(a, dtype=None, order='K', subok=True)`

Return a new array with the same shape and type as a given array.

Parameters

`a : array_like`
 The shape and data-type of `a` define these same attributes of the
 returned array.
`dtype : data-type, optional`
 Overrides the data type of the result.

.. versionadded:: 1.6.0
`order : {'C', 'F', 'A', or 'K'}, optional`
 Overrides the memory layout of the result. 'C' means C-order,
 'F' means F-order, 'A' means 'F' if ``a`` is Fortran contiguous,
 'C' otherwise. 'K' means match the layout of ``a`` as closely
 as possible.

.. versionadded:: 1.6.0
`subok : bool, optional`
 If True, then the newly created array will use the sub-class
 type of 'a', otherwise it will be a base-class array. Defaults
 to True.

Returns

`out : ndarray`
 Array of uninitialized (arbitrary) data with the same
 shape and type as `a`.

See Also

`ones_like` : Return an array of ones with shape and type of input.
`zeros_like` : Return an array of zeros with shape and type of input.
`empty` : Return a new uninitialized array.
`ones` : Return a new array setting values to one.
`zeros` : Return a new array setting values to zero.

Notes

This function does *not* initialize the returned array; to do that use
`zeros_like` or `ones_like` instead. It may be marginally faster than
the functions that do set the array values.

Examples

`>>> a = ([1,2,3], [4,5,6])` # a is array-like
`>>> np.empty_like(a)`
`array([[-1073741821, -1073741821, 3],` #random
`[0, 0, -1073741821]])`
`>>> a = np.array([[1., 2., 3.],[4.,5.,6.]])`
`>>> np.empty_like(a)`
`array([[-2.00000715e+000, 1.48219694e-323, -2.00000572e+000],#random`
`[4.38791518e-305, -2.00000715e+000, 4.17269252e-309]])`

`expand_dims(a, axis)`
Expand the shape of an array.

Insert a new axis, corresponding to a given position in the array shape.

```
Parameters
-----
a : array_like
    Input array.
axis : int
    Position (amongst axes) where new axis is to be inserted.

Returns
-----
res : ndarray
    Output array. The number of dimensions is one greater than that of
    the input array.

See Also
-----
doc.indexing, atleast_1d, atleast_2d, atleast_3d

Examples
-----
>>> x = np.array([1,2])
>>> x.shape
(2,)

The following is equivalent to ``x[np.newaxis,:,:]`` or ``x[np.newaxis]``:
>>> y = np.expand_dims(x, axis=0)
>>> y
array([[1, 2]])
>>> y.shape
(1, 2)

>>> y = np.expand_dims(x, axis=1) # Equivalent to x[:,newaxis]
>>> y
array([[1],
       [2]])
>>> y.shape
(2, 1)

Note that some examples may use ``None`` instead of ``np.newaxis``. These
are the same objects:
>>> np.newaxis is None
True

extract(condition, arr)
Return the elements of an array that satisfy some condition.

This is equivalent to ``np.compress(ravel(condition), ravel(arr))``. If
`condition` is boolean ``np.extract`` is equivalent to ``arr[condition]``.

Note that `place` does the exact opposite of `extract`.

Parameters
-----
condition : array_like
    An array whose nonzero or True entries indicate the elements of `arr`
    to extract.
arr : array_like
    Input array of the same size as `condition`.

Returns
-----
extract : ndarray
    Rank 1 array of values from `arr` where `condition` is True.

See Also
-----
take, put, copyto, compress, place

Examples
-----
>>> arr = np.arange(12).reshape((3, 4))
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> condition = np.mod(arr, 3)==0
>>> condition
array([[ True, False, False,  True],
       [False, False,  True, False],
       [False,  True, False, False]], dtype=bool)
>>> np.extract(condition, arr)
array([0, 3, 6, 9])

If `condition` is boolean:
>>> arr[condition]
```

```
array([0, 3, 6, 9])

eye(N, M=None, k=0, dtype=<type 'float'>)
    Return a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters
-----
N : int
    Number of rows in the output.
M : int, optional
    Number of columns in the output. If None, defaults to `N`.
k : int, optional
    Index of the diagonal: 0 (the default) refers to the main diagonal,
    a positive value refers to an upper diagonal, and a negative value
    to a lower diagonal.
dtype : data-type, optional
    Data-type of the returned array.

Returns
-----
I : ndarray of shape (N,M)
    An array where all elements are equal to zero, except for the `k`-th
    diagonal, whose values are equal to one.

See Also
-----
identity : (almost) equivalent function
diag : diagonal 2-D array from a 1-D array specified by the user.

Examples
-----
>>> np.eye(2, dtype=int)
array([[1, 0],
          [0, 1]])
>>> np.eye(3, k=1)
array([[ 0.,  1.,  0.],
          [ 0.,  0.,  1.],
          [ 0.,  0.,  0.]])
```

fastCopyAndTranspose = [_fastCopyAndTranspose\(...\)](#)
 [_fastCopyAndTranspose\(a\)](#)

fill_diagonal(a, val, wrap=False)
 Fill the main diagonal of the given array of any dimensionality.

For an array `a` with ``a.ndim > 2``, the diagonal is the list of locations with indices ``a[i, i, ..., i]`` all identical. This function modifies the input array in-place, it does not return a value.

Parameters

a : array, at least 2-D.
 Array whose diagonal is to be filled, it gets modified in-place.

val : scalar
 Value to be written on the diagonal, its type must be compatible with that of the array a.

wrap : bool
 For tall matrices in NumPy version up to 1.6.2, the diagonal "wrapped" after N columns. You can have this behavior with this option. This affects only tall matrices.

See also

[diag_indices](#), [diag_indices_from](#)

Notes

.. versionadded:: 1.4.0

This functionality can be obtained via `diag_indices`, but internally this version uses a much faster implementation that never constructs the indices and uses simple slicing.

Examples

>>> a = np.zeros((3, 3), [int](#))
>>> np.fill_diagonal(a, 5)
>>> a
[array](#)([[5, 0, 0],
 [0, 5, 0],
 [0, 0, 5]])

The same function can operate on a 4-D array:

>>> a = np.zeros((3, 3, 3, 3), [int](#))
>>> np.fill_diagonal(a, 4)

We only show a few blocks for clarity:

```
>>> a[0, 0]
array([[4, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> a[1, 1]
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 0]])
>>> a[2, 2]
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 4]])
```

The wrap option affects only tall matrices:

```
>>> # tall matrices no wrap
>>> a = np.zeros((5, 3), int)
>>> fill_diagonal(a, 4)
>>> a
array([[4, 0, 0],
       [0, 4, 0],
       [0, 0, 4],
       [0, 0, 0],
       [0, 0, 0]])

>>> # tall matrices wrap
>>> a = np.zeros((5, 3), int)
>>> fill_diagonal(a, 4, wrap=True)
>>> a
array([[4, 0, 0],
       [0, 4, 0],
       [0, 0, 4],
       [0, 0, 0],
       [4, 0, 0]])

>>> # wide matrices
>>> a = np.zeros((3, 5), int)
>>> fill_diagonal(a, 4, wrap=True)
>>> a
array([[4, 0, 0, 0, 0],
       [0, 4, 0, 0, 0],
       [0, 0, 4, 0, 0]])
```

find_common_type(array_types, scalar_types)

Determine common type following standard coercion rules.

Parameters

array_types : sequence
A list of dtypes or [dtype](#) convertible objects representing arrays.
scalar_types : sequence
A list of dtypes or [dtype](#) convertible objects representing scalars.

Returns

datatype : [dtype](#)
The common data type, which is the maximum of `array_types` ignoring
'scalar_types', unless the maximum of 'scalar_types' is of a
different kind ([dtype.kind](#)). If the kind is not understood, then
None is returned.

See Also

[dtype](#), common_type, can_cast, mastypecode

Examples

```
>>> np.find_common_type([], [np.int64, np.float32, np.complex])
dtype('complex128')
>>> np.find_common_type([np.int64, np.float32], [])
dtype('float64')
```

The standard casting rules ensure that a scalar cannot up-cast an array unless the scalar is of a fundamentally different kind of data (i.e. under a different hierarchy in the data type hierarchy) then the array:

```
>>> np.find_common_type([np.float32], [np.int64, np.float64])
dtype('float32')
```

Complex is of a different type, so it up-casts the [float](#) in the 'array_types' argument:

```
>>> np.find_common_type([np.float32], [np.complex])
dtype('complex128')
```

```
Type specifier strings are convertible to dtypes and can therefore
be used instead of dtypes:

>>> np.find_common_type(['f4', 'f4', 'i4'], ['c8'])
dtype('complex128')

fix(x, y=None)
    Round to nearest integer towards zero.

    Round an array of floats element-wise to nearest integer towards zero.
    The rounded values are returned as floats.

Parameters
-----
x : array\_like
    An array of floats to be rounded
y : ndarray, optional
    Output array

Returns
-----
out : ndarray of floats
    The array of rounded numbers

See Also
-----
trunc, floor, ceil
around : Round to given number of decimals

Examples
-----
>>> np.fix(3.14)
3.0
>>> np.fix(3)
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9])
array([ 2.,  2., -2., -2.])

flatnonzero(a)
    Return indices that are non-zero in the flattened version of a.

    This is equivalent to a.ravel\(\).nonzero()[0].

Parameters
-----
a : ndarray
    Input array.

Returns
-----
res : ndarray
    Output array, containing the indices of the elements of `a.ravel\(\)`
    that are non-zero.

See Also
-----
nonzero : Return the indices of the non-zero elements of the input array.
ravel : Return a 1-D array containing the elements of the input array.

Examples
-----
>>> x = np.arange(-2, 3)
>>> x
array([-2, -1, 0, 1, 2])
>>> np.flatnonzero(x)
array([0, 1, 3, 4])

    Use the indices of the non-zero elements as an index array to extract
    these elements:

>>> x.ravel()[np.flatnonzero(x)]
array([-2, -1, 1, 2])

flipr(m)
    Flip array in the left/right direction.

    Flip the entries in each row in the left/right direction.
    Columns are preserved, but appear in a different order than before.

Parameters
-----
m : array\_like
    Input array, must be at least 2-D.

Returns
-----
f : ndarray
    A view of `m` with the columns reversed. Since a view
```

```
    is returned, this operation is :math:`\mathcal{O}(1)`.

See Also
-----
flipud : Flip array in the up/down direction.
rot90 : Rotate array counterclockwise.

Notes
-----
Equivalent to A[:,::-1]. Requires the array to be at least 2-D.

Examples
-----
>>> A = np.diag([1.,2.,3.])
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
>>> np.flipud(A)
array([[ 0.,  0.,  1.],
       [ 0.,  2.,  0.],
       [ 3.,  0.,  0.]))

>>> A = np.random.randn(2,3,5)
>>> np.all(np.flipud(A)==A[:,::-1,...])
True

flipud(m)
Flip array in the up/down direction.

Flip the entries in each column in the up/down direction.
Rows are preserved, but appear in a different order than before.

Parameters
-----
m : array_like
    Input array.

Returns
-----
out : array_like
    A view of `m` with the rows reversed. Since a view is
    returned, this operation is :math:`\mathcal{O}(1)`.

See Also
-----
fliplr : Flip array in the left/right direction.
rot90 : Rotate array counterclockwise.

Notes
-----
Equivalent to ``A[::-1,...]``.
Does not require the array to be two-dimensional.

Examples
-----
>>> A = np.diag([1.0, 2, 3])
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
>>> np.flipud(A)
array([[ 0.,  0.,  3.],
       [ 0.,  2.,  0.],
       [ 1.,  0.,  0.]))

>>> A = np.random.randn(2,3,5)
>>> np.all(np.flipud(A)==A[::-1,...])
True

>>> np.flipud([1,2])
array([2, 1])

frombuffer(...)
frombuffer(buffer, dtype=float, count=-1, offset=0)

Interpret a buffer as a 1-dimensional array.

Parameters
-----
buffer : buffer_like
    An object that exposes the buffer interface.
dtype : data-type, optional
    Data-type of the returned array; default: float.
count : int, optional
    Number of items to read. ``-1`` means all data in the buffer.
offset : int, optional
    Start reading the buffer from this offset; default: 0.
```

Notes

If the buffer has data that is not in machine [byte](#)-order, this should be specified as part of the data-type, e.g.::

```
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['h', 'e', 'l', 'l', 'o'],
      dtype='|S1')
```

fromfile(...)

```
fromfile(file, dtype=float, count=-1, sep='')
```

Construct an array from data in a text or binary file.

A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the `tofile` method can be read using this function.

Parameters

file : file or [str](#)
Open file [object](#) or filename.
dtype : data-type
Data type of the returned array.
For binary files, it is used to determine the size and [byte](#)-order of the items in the file.
count : [int](#)
Number of items to read. ``-1`` means all items (i.e., the complete file).
sep : [str](#)
Separator between items if file is a text file.
Empty ("") separator means the file should be treated as binary.
Spaces (" ") in the separator match zero or more whitespace characters.
A separator consisting only of spaces must match at least one whitespace.

See also

[load](#), [save](#)
[ndarray](#).tofile
[loadtxt](#) : More [flexible](#) way of loading data from a text file.

Notes

Do not rely on the combination of `tofile` and `fromfile` for data storage, as the binary files generated are not platform independent. In particular, no [byte](#)-order or data-type information is saved. Data can be stored in the platform independent ``.npy`` format using `save` and `load` instead.

Examples

Construct an [ndarray](#):

```
>>> dt = np.dtype([('time', [('min', int), ('sec', int)]),
...                 ('temp', float)])
>>> x = np.zeros((1,), dtype=dt)
>>> x['time']['min'] = 10; x['temp'] = 98.25
>>> x
array([(10, 0), 98.25],
      dtype=[('time', [('min', 'i4'\), \('sec', 'i4'\\)\\]\\), \\('temp', 'f8'\\\)\\\)\\\]\\\)
```

Save the raw data to disk:

```
>>> import os
>>> fname = os.tmpnam()
>>> x.tofile(fname)
```

Read the raw data from disk:

```
>>> np.fromfile(fname, dtype=dt)
array([(10, 0), 98.25],
      dtype=[('time', [('min', 'i4'\), \('sec', 'i4'\\)\\]\\), \\('temp', 'f8'\\\)\\\)\\\]\\\)
```

The recommended way to store and load data:

```
>>> np.save(fname, x)
>>> np.load(fname + '.npy')
```

```

array([(10, 0), 98.25]),
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])

fromfunction(function, shape, **kwargs)
Construct an array by executing a function over each coordinate.

The resulting array therefore has a value ``fn(x, y, z)`` at
coordinate ``(x, y, z)``.

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

Returns
-----
fromfunction : any
    The result of the call to `function` is passed back directly.
    Therefore the shape of `fromfunction` is completely determined by
    `function`. If `function` returns a scalar value, the shape of
    `fromfunction` would match the `shape` parameter.

See Also
-----
indices, meshgrid

Notes
-----
Keywords other than `dtype` are passed to `function`.

Examples
-----
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

fromiter(...)
fromiter(iterable, dtype, count=-1)

Create a new 1-dimensional array from an iterable object.

Parameters
-----
iterable : iterable object
    An iterable object providing data for the array.
dtype : data-type
    The data-type of the returned array.
count : int, optional
    The number of items to read from *iterable*. The default is -1,
    which means all data is read.

Returns
-----
out : ndarray
    The output array.

Notes
-----
Specify `count` to improve performance. It allows ``fromiter`` to
pre-allocate the output array, instead of resizing it on demand.

Examples
-----
>>> iterable = (x*x for x in range(5))
>>> np.fromiter(iterable, np.float)
array([ 0.,  1.,  4.,  9., 16.])

frompyfunc(...)
frompyfunc(func, nin, nout)

Takes an arbitrary Python function and returns a Numpy ufunc.

```

Can be used, for example, to add broadcasting to a built-in Python function (see Examples section).

Parameters

func : Python function [object](#)
An arbitrary Python function.

nin : [int](#)
The [number](#) of input arguments.

nout : [int](#)
The [number](#) of objects returned by `func`.

Returns

out : [ufunc](#)
Returns a Numpy universal function (``[ufunc` ``\) \[object\]\(#\).](#)

Notes

The returned [ufunc](#) always returns [PyObject](#) arrays.

Examples

Use [frompyfunc](#) to add broadcasting to the Python function ``oct``:

```
>>> oct_array = np.frompyfunc(oct, 1, 1)
>>> oct_array(np.array((10, 30, 100)))
array([012, 036, 0144], dtype=object)
>>> np.array((oct(10), oct(30), oct(100))) # for comparison
array(['012', '036', '0144'],
      dtype='|S4')
```

fromregex(file, regexp, dtype)

Construct an array from a text file, using regular expression parsing.

The returned array is always a structured array, and is constructed from all matches of the regular expression in the file. Groups in the regular expression are converted to fields of the structured array.

Parameters

file : [str](#) or file
File name or file [object](#) to read.

regexp : [str](#) or regexp
Regular expression used to parse the file.

Groups in the regular expression correspond to fields in the [dtype](#).

[dtype](#) : [dtype](#) or list of dtypes
Dtype for the structured array.

Returns

output : [ndarray](#)
The output array, containing the part of the content of `file` that was matched by `regexp`. `output` is always a structured array.

Raises

[TypeError](#)
When `dtype` is not a valid [dtype](#) for a structured array.

See Also

[fromstring](#), [loadtxt](#)

Notes

Dtypes for structured arrays can be specified in several forms, but all forms specify at least the data type and field name. For details see [`doc.structured_arrays`](#).

Examples

```
>>> f = open('test.dat', 'w')
>>> f.write("1312 foo\n1534 bar\n444 qux")
>>> f.close()

>>> regexp = r"(\d+)\s+(...)" # match [digits, whitespace, anything]
>>> output = np.fromregex('test.dat', regexp,
...                         [('num', np.int64), ('key', 'S3')])
>>> output
array([(1312L, 'foo'), (1534L, 'bar'), (444L, 'qux')],
      dtype=\[\('num', '<i8'\), \('key', '|S3'\)\]\)\]\)
>>> output\['num'\]
array\(\[1312, 1534, 444\], dtype=int64\\)
```

fromstring(...)

[fromstring](#)(string, [dtype=float](#), count=-1, sep='')

A new 1-D array initialized from raw binary or text data in a string.

Parameters

`string : str`
A string containing the data.

`dtype : data-type, optional`
The data type of the array; default: `float`. For binary input data, the data must be in exactly this format.

`count : int, optional`
Read this `number` of ``dtype`` elements from the data. If this is negative (the default), the count will be determined from the length of the data.

`sep : str, optional`
If not provided or, equivalently, the empty string, the data will be interpreted as binary data; otherwise, as ASCII text with decimal numbers. Also in this latter case, this argument is interpreted as the string separating numbers in the data; extra whitespace between elements is also ignored.

Returns

`arr : ndarray`
The constructed array.

Raises

`ValueError`
If the string is not the correct size to satisfy the requested `dtype` and `count`.

See Also

`frombuffer`, `fromfile`, `fromiter`

Examples

```
>>> np.fromstring('\x01\x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
>>> np.fromstring('\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
```

full(shape, fill_value, dtype=None, order='C')
Return a new array of given shape and type, filled with `fill_value`.

Parameters

`shape : int or sequence of ints`
Shape of the new array, e.g., ``(2, 3)`` or ``2``.

`fill_value : scalar`
Fill value.

`dtype : data-type, optional`
The desired data-type for the array, e.g., `np.int8`. Default is `float`, but will change to `np.array(fill_value).dtype` in a future release.

`order : {'C', 'F'}, optional`
Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns

`out : ndarray`
Array of `fill_value` with the given shape, `dtype`, and order.

See Also

`zeros_like` : Return an array of zeros with shape and type of input.

`ones_like` : Return an array of ones with shape and type of input.

`empty_like` : Return an empty array with shape and type of input.

`full_like` : Fill an array with shape and type of input.

`zeros` : Return a new array setting values to zero.

`ones` : Return a new array setting values to one.

`empty` : Return a new uninitialized array.

Examples

```
>>> np.full((2, 2), np.inf)
array([[ inf,  inf],
       [ inf,  inf]])
>>> np.full((2, 2), 10, dtype=np.int)
array([[10, 10],
       [10, 10]])
```

full_like(a, fill_value, dtype=None, order='K', subok=True)

Return a full array with the same shape and type as a given array.

Parameters

a : array_like
The shape and data-type of `a` define these same attributes of the returned array.

fill_value : scalar
Fill value.

dtype : data-type, optional
Overrides the data type of the result.

order : {'C', 'F', 'A', or 'K'}, optional
Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.

subok : bool, optional.
If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

Returns

out : ndarray
Array of `fill_value` with the same shape and type as `a`.

See Also

zeros_like : Return an array of zeros with shape and type of input.
ones_like : Return an array of ones with shape and type of input.
empty_like : Return an empty array with shape and type of input.
zeros : Return a new array setting values to zero.
ones : Return a new array setting values to one.
empty : Return a new uninitialized array.
full : Fill a new array.

Examples

```
>>> x = np.arange(6, dtype=np.int)
>>> np.full_like(x, 1)
array([1, 1, 1, 1, 1, 1])
>>> np.full_like(x, 0.1)
array([ 0. ,  0.1,  0. ,  0.1,  0. ,  0.1])
>>> np.full_like(x, np.nan, dtype=np.double)
array([ nan,  nan,  nan,  nan,  nan,  nan])
```

```
>>> y = np.arange(6, dtype=np.double)
>>> np.full_like(y, 0.1)
array([ 0.1,  0.1,  0.1,  0.1,  0.1,  0.1])
```

fv(rate, nper, pmt, pv, when='end')
Compute the future value.

Given:
 * a present value, `pv`
 * an interest `rate` compounded once per period, of which there are
 * `nper` total
 * a (fixed) payment, `pmt`, paid either
 * at the beginning (`when` = {'begin', 1}) or the end (`when` = {'end', 0}) of each period

Return:
the value at the end of the `nper` periods

Parameters

rate : scalar or array_like of **shape**(M,)
Rate of interest as decimal (not per cent) per period

nper : scalar or array_like of **shape**(M,)
Number of compounding periods

pmt : scalar or array_like of **shape**(M,)
Payment

pv : scalar or array_like of **shape**(M,)
Present value

when : {{'begin', 1}, {'end', 0}}, {string, **int**}, optional
When payments are due ('begin' (1) or 'end' (0)).
Defaults to {'end', 0}.

Returns

out : ndarray
Future values. If all input is scalar, returns a scalar **float**. If any input is array_like, returns future values for each input element. If multiple inputs are array_like, they all must have the same shape.

Notes

```
-----
The future value is computed by solving the equation:::

fv +
pv*(1+rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) == 0

or, when ``rate == 0``::

fv + pv + pmt * nper == 0

References
-----
.. [WRW] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May).
   Open Document Format for Office Applications (OpenDocument)v1.2,
   Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version,
   Pre-Draft 12. Organization for the Advancement of Structured Information
   Standards (OASIS). Billerica, MA, USA. [ODT Document].
Available:
http://www.oasis-open.org/committees/documents.php?wg\_abbrev=office-formula
OpenDocument-formula-20090508.odt

Examples
-----
What is the future value after 10 years of saving $100 now, with
an additional monthly savings of $100. Assume the interest rate is
5% (annually) compounded monthly?

>>> np.fv(0.05/12, 10*12, -100, -100)
15692.928894335748

By convention, the negative sign represents cash flow out (i.e. money not
available today). Thus, saving $100 a month at 5% annual interest leads
to $15,692.93 available to spend in 10 years.

If any input is array_like, returns an array of equal shape. Let's
compare different interest rates from the example above.

>>> a = np.array((0.05, 0.06, 0.07))/12
>>> np.fv(a, 10*12, -100, -100)
array([ 15692.92889434, 16569.87435405, 17509.44688102])

genfromtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, skip_header=0, skip_footer=0,
converters=None, missing_values=None, filling_values=None, usecols=None, names=None, excludelist=None,
deletechars=None, replace_space='_', autostrip=False, case_sensitive=True, defaultfmt='f%i', unpack=None,
usemask=False, loose=True, invalid_raise=True, max_rows=None)
Load data from a text file, with missing values handled as specified.

Each line past the first `skip_header` lines is split at the `delimiter`
character, and characters following the `comments` character are discarded.

Parameters
-----
fname : file, str, list of str, generator
   File, filename, list, or generator to read. If the filename
   extension is '.gz' or '.bz2', the file is first decompressed. Note
   that generators must return byte strings in Python 3k. The strings
   in a list or produced by a generator are treated as lines.
dtype : dtype, optional
   Data type of the resulting array.
   If None, the dtypes will be determined by the contents of each
   column, individually.
comments : str, optional
   The character used to indicate the start of a comment.
   All the characters occurring on a line after a comment are discarded
delimiter : str, int, or sequence, optional
   The string used to separate values. By default, any consecutive
   whitespaces act as delimiter. An integer or sequence of integers
   can also be provided as width(s) of each field.
skiprows : int, optional
   `skiprows` was removed in numpy 1.10. Please use `skip_header` instead.
skip_header : int, optional
   The number of lines to skip at the beginning of the file.
skip_footer : int, optional
   The number of lines to skip at the end of the file.
converters : variable, optional
   The set of functions that convert the data of a column to a value.
   The converters can also be used to provide a default value
   for missing data: ``converters = {3: lambda s: float(s or 0)}``.
missing : variable, optional
   'missing' was removed in numpy 1.10. Please use 'missing_values'
   instead.
missing_values : variable, optional
   The set of strings corresponding to missing data.
filling_values : variable, optional
   The set of values to be used as default when the data are missing.
usecols : sequence, optional
   Which columns to read, with 0 being the first. For example,
```

```

``usecols = (1, 4, 5)`` will extract the 2nd, 5th and 6th columns.
names : {None, True, str, sequence}, optional
    If `names` is True, the field names are read from the first valid line
    after the first `skip_header` lines.
    If `names` is a sequence or a single string of comma-separated names,
    the names will be used to define the field names in a structured dtype.
    If `names` is None, the names of the dtype fields will be used, if any.
excludelist : sequence, optional
    A list of names to exclude. This list is appended to the default list
    ['return', 'file', 'print']. Excluded names are appended an underscore:
    for example, 'file' would become 'file_'.
deletechars : str, optional
    A string combining invalid characters that must be deleted from the
    names.
defaultfmt : str, optional
    A format used to define default field names, such as "f%i" or "f_%02i".
autostrip : bool, optional
    Whether to automatically strip white spaces from the variables.
replace_space : char, optional
    Character(s) used in replacement of white spaces in the variables
    names. By default, use a '-'.
case_sensitive : {True, False, 'upper', 'lower'}, optional
    If True, field names are case sensitive.
    If False or 'upper', field names are converted to upper case.
    If 'lower', field names are converted to lower case.
unpack : bool, optional
    If True, the returned array is transposed, so that arguments may be
    unpacked using ``x, y, z = loadtxt(...)``.
usemask : bool, optional
    If True, return a masked array.
    If False, return a regular array.
loose : bool, optional
    If True, do not raise errors for invalid values.
invalid_raise : bool, optional
    If True, an exception is raised if an inconsistency is detected in the
    number of columns.
    If False, a warning is emitted and the offending lines are skipped.
max_rows : int, optional
    The maximum number of rows to read. Must not be used with skip_footer
    at the same time. If given, the value must be at least 1. Default is
    to read the entire file.

.. versionadded:: 1.10.0

```

Returns

out : [ndarray](#)
 Data read from the text file. If `usemask` is True, this is a
 masked array.

See Also

[numpy.loadtxt](#) : equivalent function when no data is missing.

Notes

- * When spaces are used as delimiters, or when no delimiter has been given
 as input, there should not be any missing data between two fields.
 * When the variables are named (either by a [flexible dtype](#) or with `names`),
 there must not be any header in the file (else a [ValueError](#)
exception is raised).
 * Individual values are not stripped of spaces by default.
 When using a custom converter, make sure the function does remove spaces.

References

 .. [1] Numpy User Guide, section `I/O with Numpy
[<http://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html>](http://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html).

Examples

`>>> from io import StringIO`
`>>> import numpy as np`

Comma delimited file with mixed [dtype](#)

```

>>> s = StringIO("1,1.3,abcde")
>>> data = np.genfromtxt(s, dtype=[('myint','i8'),('myfloat','f8'),
... ('mystring','S5')], delimiter=",")
>>> data
array((1, 1.3, 'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', '|S5')])

```

Using [dtype](#) = None

```

>>> s.seek(0) # needed for StringIO example only
>>> data = np.genfromtxt(s, dtype=None,
... names = ['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, 'abcde'),

```

```
        dtype=[('myint', 'i8'), ('myfloat', 'f8'), ('mystring', '|S5')])
```

Specifying **dtype** and names

```
>>> s.seek(0)
>>> data = np.genfromtxt(s, dtype="i8,f8,S5",
...     names=['myint','myfloat','mystring'], delimiter=",")
>>> data
array([(1, 1.3, 'abcde'),
       dtype=[('myint', 'i8'), ('myfloat', 'f8'), ('mystring', '|S5')])
```

An example with fixed-width columns

```
>>> s = StringIO("11.3abcde")
>>> data = np.genfromtxt(s, dtype=None, names=['intvar','fltvvar','strvar'],
...     delimiter=[1,3,5])
>>> data
array([(1, 1.3, 'abcde'),
       dtype=[('intvar', 'i8'), ('fltvvar', 'f8'), ('strvar', '|S5')])
```

get_array_wrap(*args)

Find the wrapper for the array with the highest priority.

In case of ties, leftmost wins. If no wrapper is found, return None

get_include()

Return the directory that contains the NumPy *.h header files.

Extension modules that need to compile against NumPy should use this function to locate the appropriate include directory.

Notes

When using ``distutils``, for example in ``setup.py``.

```
:::
```

```
import numpy as np
...
Extension('extension_name', ...
          include_dirs=[np.get_include()])
...
```

get_printoptions()

Return the current print options.

Returns

print_opts : dict

Dictionary of current print options with keys

```
- precision : int
- threshold : int
- edgeitems : int
- linewidth : int
- suppress : bool
- nanstr : str
- infstr : str
- formatter : dict of callables
```

For a full description of these options, see `set_printoptions`.

See Also

set_printoptions, **set_string_function**

getbuffer(...)

```
getbuffer(obj [,offset[, size]])
```

Create a buffer **object** from the given **object** referencing a slice of length **size** starting at **offset**.

Default is the entire buffer. A read-write buffer is attempted followed by a read-only buffer.

Parameters

obj : **object**

offset : **int**, optional

size : **int**, optional

Returns

buffer_obj : buffer

Examples

```
-----
>>> buf = np.getbuffer(np.ones(5), 1, 3)
>>> len(buf)
3
>>> buf[0]
'\x00'
>>> buf
<read-write buffer for 0x8af1e70, size 3, offset 1 at 0x8ba4ec0>

getbufsize()
Return the size of the buffer used in ufuncs.

>Returns
-----
getbufsize : int
    Size of ufunc buffer in bytes.

geterr()
Get the current way of handling floating-point errors.

>Returns
-----
res : dict
    A dictionary with keys "divide", "over", "under", and "invalid",
    whose values are from the strings "ignore", "print", "log", "warn",
    "raise", and "call". The keys represent possible floating-point
    exceptions, and the values define how these exceptions are handled.

See Also
-----
geterrcall, seterr, seterrcall

Notes
-----
For complete documentation of the types of floating-point exceptions and
treatment options, see `seterr`.

Examples
-----
>>> np.geterr()
{'over': 'warn', 'divide': 'warn', 'invalid': 'warn',
'under': 'ignore'}
>>> np.arange(3.) / np.arange(3.)
array([ NaN, 1., 1.])

>>> oldsettings = np.seterr(all='warn', over='raise')
>>> np.geterr()
{'over': 'raise', 'divide': 'warn', 'invalid': 'warn', 'under': 'warn'}
>>> np.arange(3.) / np.arange(3.)
__main__:1: RuntimeWarning: invalid value encountered in divide
array([ NaN, 1., 1.])

geterrcall()
Return the current callback function used on floating-point errors.

When the error handling for a floating-point error (one of "divide",
"over", "under", or "invalid") is set to 'call' or 'log', the function
that is called or the log instance that is written to is returned by
`geterrcall`. This function or log instance has been set with
`seterrcall`.

>Returns
-----
errobj : callable, log instance or None
    The current error handler. If no handler was set through `seterrcall`,
    ``None`` is returned.

See Also
-----
seterrcall, seterr, geterr

Notes
-----
For complete documentation of the types of floating-point exceptions and
treatment options, see `seterr`.

Examples
-----
>>> np.geterrcall() # we did not yet set a handler, returns None

>>> oldsettings = np.seterr(all='call')
>>> def err_handler(type, flag):
...     print("Floating point error (%s), with flag %s" % (type, flag))
>>> oldhandler = np.seterrcall(err_handler)
>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([ Inf, Inf, Inf])

```

```

>>> cur_handler = np.geterrcall()
>>> cur_handler is err_handler
True

geterrobj(...)
geterrobj()

Return the current object that defines floating-point error handling.

The error object contains all information that defines the error handling
behavior in Numpy. `geterrobj` is used internally by the other
functions that get and set error handling behavior (`geterr`, `seterr`,
`geterrcall`, `seterrcall`).

>Returns
-----
errobj : list
    The error object, a list containing three elements:
    [internal numpy buffer size, error mask, error callback function].

    The error mask is a single integer that holds the treatment information
    on all four floating point errors. The information for each error type
    is contained in three bits of the integer. If we print it in base 8, we
    can see what treatment is set for "invalid", "under", "over", and
    "divide" (in that order). The printed string can be interpreted with

    * 0 : 'ignore'
    * 1 : 'warn'
    * 2 : 'raise'
    * 3 : 'call'
    * 4 : 'print'
    * 5 : 'log'

See Also
-----
seterrobj, seterr, geterr, seterrcall, geterrcall
getbufsize, setbufsize

Notes
-----
For complete documentation of the types of floating-point exceptions and
treatment options, see `seterr`.

Examples
-----
>>> np.geterrobj() # first get the defaults
[10000, 0, None]

>>> def err_handler(type, flag):
...     print("Floating point error (%s), with flag %s" % (type, flag))
...
>>> old_bufsize = np.setbufsize(20000)
>>> old_err = np.seterr(divide='raise')
>>> old_handler = np.seterrcall(err_handler)
>>> np.geterrobj()
[20000, 2, <function err_handler at 0x91dcaac>]

>>> old_err = np.seterr(all='ignore')
>>> np.base_repr(np.geterrobj()[1], 8)
'0'
>>> old_err = np.seterr(divide='warn', over='log', under='call',
...                      invalid='print')
>>> np.base_repr(np.geterrobj()[1], 8)
'4351'

gradient(f, *varargs, **kwargs)
Return the gradient of an N-dimensional array.

The gradient is computed using second order accurate central differences
in the interior and either first differences or second order accurate
one-sides (forward or backwards) differences at the boundaries. The
returned gradient hence has the same shape as the input array.

Parameters
-----
f : array_like
    An N-dimensional array containing samples of a scalar function.
varargs : scalar or list of scalar, optional
    N scalars specifying the sample distances for each dimension,
    i.e. `dx`, `dy`, `dz`, ... Default distance: 1.
    single scalar specifies sample distance for all dimensions.
    if `axis` is given, the number of varargs must equal the number of axes.
edge_order : {1, 2}, optional
    Gradient is calculated using Nth order accurate differences
    at the boundaries. Default: 1.
.. versionadded:: 1.9.1

```

```

axis : None or int or tuple of ints, optional
    Gradient is calculated only along the given axis or axes
    The default (axis = None) is to calculate the gradient for all the axes of the input array.
    axis may be negative, in which case it counts from the last to the first axis.

.. versionadded:: 1.11.0

>Returns
-----
gradient : list of ndarray
    Each element of `list` has the same shape as `f` giving the derivative
    of `f` with respect to each dimension.

>Examples
-----
>>> x = np.array([1, 2, 4, 7, 11, 16], dtype=np.float)
>>> np.gradient(x)
array\(\[ 1.,  1.5,  2.5,  3.5,  4.5,  5. \]\)
>>> np.gradient(x, 2)
array\(\[ 0.5,  0.75,  1.25,  1.75,  2.25,  2.5 \]\)

For two dimensional arrays, the return will be two arrays ordered by
axis. In this example the first array stands for the gradient in
rows and the second one in columns direction:

>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]]), dtype=np.float)
\[array\(\[\[ 2.,  2., -1.\],
       \[ 2.,  2., -1.\]\]\), array\(\[\[ 1.,  2.5,  4.\],
       \[ 1.,  1.,  1.\]\]\)\]

>>> x = np.array([0, 1, 2, 3, 4])
>>> dx = np.gradient(x)
>>> y = x**2
>>> np.gradient(y, dx, edge_order=2)
array\(\[-0.,  2.,  4.,  6.,  8.\]\)

The axis keyword can be used to specify a subset of axes of which the gradient is calculated
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]]), dtype=np.float, axis=0)
array\(\[\[ 2.,  2., -1.\],
       \[ 2.,  2., -1.\]\]\)

```

hamming(M)

Return the Hamming window.

The Hamming window is a taper formed by using a weighted cosine.

Parameters

M : [int](#)
Number of points in the output window. If zero or less, an
empty array is returned.

Returns

out : [ndarray](#)
The window, with the maximum value normalized to one (the value
one appears only if the [number](#) of samples is odd).

See Also

[bartlett](#), [blackman](#), [hanning](#), [kaiser](#)

Notes

The Hamming window is defined as

$$\dots \text{math}:: w(n) = 0.54 - 0.46\cos(\frac{2\pi n}{M-1}) \quad \text{for } 0 \leq n \leq M-1$$

The Hamming was named for R. W. Hamming, an associate of J. W. Tukey and is described in Blackman and Tukey. It was recommended for smoothing the truncated autocovariance function in the time domain. Most references to the Hamming window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

- .. [1] Blackman, R.B. and Tukey, J.W., (1958) *The measurement of power spectra*, Dover Publications, New York.
- .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.
- .. [3] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function
- .. [4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 425.

```

Examples
-----
>>> np.hamming(12)
array([ 0.08      ,  0.15302337,  0.34890909,  0.60546483,  0.84123594,
       0.98136677,  0.98136677,  0.84123594,  0.60546483,  0.34890909,
       0.15302337,  0.08      ])

```

Plot the window and the frequency response:

```

>>> from numpy.fft import fft, fftshift
>>> window = np.hamming(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Hamming window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Hamming window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()

```

hamming(M)

Return the Hanning window.

The Hanning window is a taper formed by using a weighted cosine.

Parameters

M : **int**
 Number of points in the output window. If zero or less, an empty array is returned.

Returns

out : **ndarray**, **shape**(M,)
 The window, with the maximum value normalized to one (the value one appears only if `M` is odd).

See Also

bartlett, blackman, hamming, kaiser

Notes

The Hanning window is defined as

$$\dots \text{math:: } w(n) = 0.5 - 0.5 \cos \left(\frac{2\pi n}{M-1} \right) \quad \forall n \in \{0, \dots, M-1\}$$

The Hanning was named for Julius von Hann, an Austrian meteorologist. It is also known as the Cosine Bell. Some authors prefer that it be called a Hann window, to help avoid confusion with the very similar Hamming window.

Most references to the Hanning window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

- .. [1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.
- .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 106-108.
- .. [3] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function
- .. [4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 425.

Examples

```
>>> -----
>>> np.hanning(12)
array([ 0.          ,  0.07937323,  0.29229249,  0.57115742,  0.82743037,
       0.97974649,  0.97974649,  0.82743037,  0.57115742,  0.29229249,
       0.07937323,  0.          ])
```

Plot the window and its frequency response:

```
>>> from numpy.fft import fft, fftshift
>>> window = np.hanning(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Hann window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of the Hann window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()
```

histogram(a, bins=10, range=None, normed=False, weights=None, density=None)

Compute the histogram of a set of data.

Parameters

```
-----
```

a : array_like
Input data. The histogram is computed over the flattened array.

bins : int or sequence of scalars or str, optional
If `bins` is an int, it defines the number of equal-width bins in the given range (10, by default). If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

.. versionadded:: 1.11.0
If `bins` is a string from the list below, `histogram` will use the method chosen to calculate the optimal bin width and consequently the number of bins (see `Notes` for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the 'auto' option is suggested. Weighted data is not supported for automated bin size selection.

'auto'
Maximum of the 'sturges' and 'fd' estimators. Provides good all round performance

'fd' (Freedman Diaconis Estimator)
Robust (resilient to outliers) estimator that takes into account data variability and data size .

'doane'
An improved version of Sturges' estimator that works better with non-normal datasets.

'scott'
Less robust estimator that that takes into account data variability and data size.

'rice'
Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.

'sturges'
R's default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins

```

    for large non-gaussian datasets.

'sqrt'
    Square root (of data size) estimator, used by Excel and
    other programs for its speed and simplicity.

range : (float, float), optional
    The lower and upper range of the bins. If not provided, range
    is simply ``a.min(), a.max()``. Values outside the range are
    ignored. The first element of the range must be less than or
    equal to the second. `range` affects the automatic bin
    computation as well. While bin width is computed to be optimal
    based on the actual data within `range`, the bin count will fill
    the entire range including portions containing no data.

normed : bool, optional
    This keyword is deprecated in Numpy 1.6 due to confusing/buggy
    behavior. It will be removed in Numpy 2.0. Use the ``density``
    keyword instead. If ``False``, the result will contain the
    number of samples in each bin. If ``True``, the result is the
    value of the probability *density* function at the bin,
    normalized such that the *integral* over the range is 1. Note
    that this latter behavior is known to be buggy with unequal bin
    widths; use ``density`` instead.

weights : array_like, optional
    An array of weights, of the same shape as `a`. Each value in
    `a` only contributes its associated weight towards the bin count
    (instead of 1). If `density` is True, the weights are
    normalized, so that the integral of the density over the range
    remains 1.

density : bool, optional
    If ``False``, the result will contain the number of samples in
    each bin. If ``True``, the result is the value of the
    probability *density* function at the bin, normalized such that
    the *integral* over the range is 1. Note that the sum of the
    histogram values will not be equal to 1 unless bins of unity
    width are chosen; it is not a probability *mass* function.

    Overrides the ``normed`` keyword if given.

Returns
-----
hist : array
    The values of the histogram. See `density` and `weights` for a
    description of the possible semantics.
bin_edges : array of dtype float
    Return the bin edges ``(length(hist)+1)``.

See Also
-----
histogramdd, bincount, searchsorted, digitize

Notes
-----
All but the last (righthand-most) bin is half-open. In other words,
if `bins` is::

    [1, 2, 3, 4]

then the first bin is ``[1, 2)`` (including 1, but excluding 2) and
the second ``[2, 3)``. The last bin, however, is ``[3, 4)``, which
*includes* 4.

.. versionadded:: 1.11.0

The methods to estimate the optimal number of bins are well founded
in literature, and are inspired by the choices R provides for
histogram visualisation. Note that having the number of bins
proportional to  $n^{1/3}$  is asymptotically optimal, which is
why it appears in most estimators. These are simply plug-in methods
that give good starting points for number of bins. In the equations
below,  $h$  is the binwidth and  $n_h$  is the number of
bins. All estimators that compute bin counts are recast to bin width
using the `ptp` of the data. The final bin count is obtained from
``np.round(np.ceil(range / h))``.

'Auto' (maximum of the 'Sturges' and 'FD' estimators)
A compromise to get a good value. For small datasets the Sturges
value will usually be chosen, while larger datasets will usually
default to FD. Avoids the overly conservative behaviour of FD
and Sturges for small and large datasets respectively.
Switchover point is usually  $a.size \approx 1000$ .

'FD' (Freedman Diaconis Estimator)
.. math:: h = 2 \frac{\text{IQR}}{n^{1/3}}

The binwidth is proportional to the interquartile range (IQR)
and inversely proportional to cube root of  $a.size$ . Can be too
conservative for small datasets, but is quite good for large
datasets. The IQR is very robust to outliers.

```

```
'Scott'
.. math:: h = \sigma \sqrt[3]{\frac{24 * \sqrt{\pi}}{n}}
```

The binwidth is proportional to the standard deviation of the data and inversely proportional to cube root of ``x.size``. Can be too conservative for small datasets, but is quite good for large datasets. The standard deviation is not very robust to outliers. Values are very similar to the Freedman-Diaconis estimator in the absence of outliers.

```
'Rice'
.. math:: n_h = 2n^{1/3}
```

The number of bins is only proportional to cube root of ``a.size``. It tends to overestimate the number of bins and it does not take into account data variability.

```
'Sturges'
.. math:: n_h = \log_2 n + 1
```

The number of bins is the base 2 log of ``a.size``. This estimator assumes normality of data and is too conservative for larger, non-normal datasets. This is the default method in R's ``hist`` method.

```
'Doane'
.. math:: n_h = 1 + \log_2(1 + \frac{|g_1|}{\sigma_{g_1}})
```

$$g_1 = \text{mean}[(x - \mu)^3]$$

$$\sigma_{g_1} = \sqrt{6(n-2)(n+1)(n+3)}$$

An improved version of Sturges' formula that produces better estimates for non-normal datasets. This estimator attempts to account for the skew of the data.

```
'Sqrt'
.. math:: n_h = \sqrt{n}
```

The simplest and fastest estimator. Only takes into account the data size.

Examples

```
-----
>>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
(array([0, 2, 1]), array([0, 1, 2, 3]))
>>> np.histogram(np.arange(4), bins=np.arange(5, density=True))
(array([ 0.25,  0.25,  0.25,  0.25]), array([0, 1, 2, 3, 4]))
>>> np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3])
(array([1, 4, 1]), array([0, 1, 2, 3]))
```

```
>>> a = np.arange(5)
>>> hist, bin_edges = np.histogram(a, density=True)
>>> hist
array([ 0.5,  0. ,  0.5,  0. ,  0.5,  0. ,  0.5,  0. ,  0.5])
>>> hist.sum()
2.4999999999999996
>>> np.sum(hist*np.diff(bin_edges))
1.0
```

.. versionadded:: 1.11.0

Automated Bin Selection Methods example, using 2 peak random data with 2000 points:

```
>>> import matplotlib.pyplot as plt
>>> rng = np.random.RandomState(10) # deterministic random data
>>> a = np.vstack((rng.normal(size=1000),
...                 rng.normal(loc=5, scale=2, size=1000)))
>>> plt.hist(a, bins='auto') # plt.hist passes its arguments to np.histogram
>>> plt.title("Histogram with 'auto' bins")
>>> plt.show()
```

histogram2d(*x*, *y*, *bins*=10, *range*=None, *normed*=False, *weights*=None)
Compute the bi-dimensional histogram of two data samples.

Parameters

```
-----
x : array_like, shape (N,)
    An array containing the x coordinates of the points to be histogrammed.
y : array_like, shape (N,)
    An array containing the y coordinates of the points to be histogrammed.
bins : int or array_like or [int, int] or [array, array], optional
    The bin specification:
```

* If int, the number of bins for the two dimensions ($n_x=n_y=bins$).

```

* If array_like, the bin edges for the two dimensions
  (x_edges=y_edges=bins).
* If [int, int], the number of bins in each dimension
  (nx, ny = bins).
* If [array, array], the bin edges in each dimension
  (x_edges, y_edges = bins).
* A combination [int, array] or [array, int], where int
  is the number of bins and array is the bin edges.

range : array_like, shape(2,2), optional
    The leftmost and rightmost edges of the bins along each dimension
    (if not specified explicitly in the 'bins' parameters):
    ``[[xmin, xmax], [ymin, ymax]]``. All values outside of this range
    will be considered outliers and not tallied in the histogram.

normed : bool, optional
    If False, returns the number of samples in each bin. If True,
    returns the bin density ``bin_count / sample_count / bin_area``.

weights : array_like, shape(N,), optional
    An array of values ``w_i`` weighing each sample `` $(x_i, y_i)$ ``.
    Weights are normalized to 1 if 'normed' is True. If 'normed' is
    False, the values of the returned histogram are equal to the sum of
    the weights belonging to the samples falling into each bin.

Returns
-----
H : ndarray, shape(nx, ny)
    The bi-dimensional histogram of samples `x` and `y`. Values in `x`
    are histogrammed along the first dimension and values in `y` are
    histogrammed along the second dimension.

xedges : ndarray, shape(nx,)
    The bin edges along the first dimension.

yedges : ndarray, shape(ny,)
    The bin edges along the second dimension.

See Also
-----
histogram : 1D histogram
histogramdd : Multidimensional histogram

Notes
-----
When 'normed' is True, then the returned histogram is the sample
density, defined such that the sum over bins of the product
``bin_value * bin_area`` is 1.

Please note that the histogram does not follow the Cartesian convention
where `x` values are on the abscissa and `y` values on the ordinate
axis. Rather, `x` is histogrammed along the first dimension of the
array (vertical), and `y` along the second dimension of the array
(horizontal). This ensures compatibility with 'histogramdd'.

Examples
-----
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt

Construct a 2D-histogram with variable bin width. First define the bin
edges:

>>> xedges = [0, 1, 1.5, 3, 5]
>>> yedges = [0, 2, 3, 4, 6]

Next we create a histogram H with random bin content:

>>> x = np.random.normal(3, 1, 100)
>>> y = np.random.normal(1, 1, 100)
>>> H, xedges, yedges = np.histogram2d(y, x, bins=(xedges, yedges))

Or we fill the histogram H with a determined bin content:

>>> H = np.ones((4, 4)).cumsum().reshape(4, 4)
>>> print(H[::-1]) # This shows the bin content in the order as plotted
[[ 13.  14.  15.  16.]
 [  9.  10.  11.  12.]
 [  5.   6.   7.   8.]
 [  1.   2.   3.   4.]]

Imshow can only do an equidistant representation of bins:

>>> fig = plt.figure(figsize=(7, 3))
>>> ax = fig.add_subplot(131)
>>> ax.set_title('imshow: equidistant')
>>> im = plt.imshow(H, interpolation='nearest', origin='low',
                    extent=[xedges[0], xedges[-1], yedges[0], yedges[-1]])

pcolormesh can display exact bin edges:

>>> ax = fig.add_subplot(132)
>>> ax.set_title('pcolormesh: exact bin edges')
>>> X, Y = np.meshgrid(xedges, yedges)

```

```

>>> ax.pcolormesh(X, Y, H)
>>> ax.set_aspect('equal')

NonUniformImage displays exact bin edges with interpolation:

>>> ax = fig.add_subplot(133)
>>> ax.set_title('NonUniformImage: interpolated')
>>> im = mpl.image.NonUniformImage(ax, interpolation='bilinear')
>>> xcenters = xedges[:-1] + 0.5 * (xedges[1:] - xedges[:-1])
>>> ycenters = yedges[:-1] + 0.5 * (yedges[1:] - yedges[:-1])
>>> im.set_data(xcenters, ycenters, H)
>>> ax.images.append(im)
>>> ax.set_xlim(xedges[0], xedges[-1])
>>> ax.set_ylim(yedges[0], yedges[-1])
>>> ax.set_aspect('equal')
>>> plt.show()

histogramdd(sample, bins=10, range=None, normed=False, weights=None)
Compute the multidimensional histogram of some data.

Parameters
-----
sample : array_like
    The data to be histogrammed. It must be an (N,D) array or data
    that can be converted to such. The rows of the resulting array
    are the coordinates of points in a D dimensional polytope.
bins : sequence or int, optional
    The bin specification:
        * A sequence of arrays describing the bin edges along each dimension.
        * The number of bins for each dimension (nx, ny, ... =bins)
        * The number of bins for all dimensions (nx=ny=...=bins).
range : sequence, optional
    A sequence of lower and upper bin edges to be used if the edges are
    not given explicitly in 'bins'. Defaults to the minimum and maximum
    values along each dimension.
normed : bool, optional
    If False, returns the number of samples in each bin. If True,
    returns the bin density ``bin_count / sample_count / bin_volume``.
weights : (N,) array_like, optional
    An array of values `w_i` weighing each sample `(x_i, y_i, z_i, ...)`.
    Weights are normalized to 1 if normed is True. If normed is False,
    the values of the returned histogram are equal to the sum of the
    weights belonging to the samples falling into each bin.

Returns
-----
H : ndarray
    The multidimensional histogram of sample x. See normed and weights
    for the different possible semantics.
edges : list
    A list of D arrays describing the bin edges for each dimension.

See Also
-----
histogram: 1-D histogram
histogram2d: 2-D histogram

Examples
-----
>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5, 8, 4), 6, 9, 5)

hsplit(ary, indices_or_sections)
Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the `split` documentation. 'hsplit' is equivalent
to `split` with `axis=1`, the array is always split along the second
axis regardless of the array dimension.

See Also
-----
split : Split an array into multiple sub-arrays of equal size.

Examples
-----
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
       [ 4.,  5.]]),

```

```

[ 8.,  9.],
[ 12., 13.]),
array([[ 2.,  3.],
[ 6.,  7.],
[ 10., 11.],
[ 14., 15.]])
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
[ 4.,  5.,  6.],
[ 8.,  9., 10.],
[ 12., 13., 14.])),
array([[ 3.],
[ 7.],
[ 11.],
[ 15.]])]
```

`array`[], `dtype=float64])`

With a higher dimensional array the split is still along the second axis.

```

>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[ 0.,  1.],
[ 2.,  3.],
[[ 4.,  5.],
[ 6.,  7.]]])
>>> np.hsplit(x, 2)
[array([[[ 0.,  1.],
[[ 4.,  5.]]]),
array([[[ 2.,  3.],
[[ 6.,  7.]]]])
```

`hstack(tup)`

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a `singl`e array. Rebuild arrays divided by 'hsplit'.

Parameters

`tup` : sequence of `ndarrays`
All arrays must have the same shape along all but the second axis.

Returns

`stacked` : `ndarray`
The array formed by stacking the given arrays.

See Also

`stack` : Join a sequence of arrays along a new axis.
`vstack` : Stack arrays in sequence vertically (row wise).
`dstack` : Stack arrays in sequence depth wise (along third axis).
`concatenate` : Join a sequence of arrays along an existing axis.
`hsplit` : Split array along second axis.

Notes

Equivalent to ```np.concatenate(tup, axis=1)```

Examples

```

>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.hstack((a,b))
array([[1, 2],
[2, 3],
[3, 4]])
```

`i0(x)`

Modified Bessel function of the first kind, order 0.

Usually denoted :math:`I_0` . This function does `broadcast`, but will *not* "up-cast" `int` `dtype` arguments unless accompanied by at least one `float` or `complex` `dtype` argument (see Raises below).

Parameters

`x` : `array_like`, `dtype float` or `complex`
Argument of the Bessel function.

Returns

`out` : `ndarray`, `shape = x.shape`, `dtype = x.dtype`
The modified Bessel function evaluated at each of the elements of `x` .

```
Raises
-----
TypeError: array cannot be safely cast to required type
    If argument consists exclusively of int dtypes.

See Also
-----
scipy.special.iv, scipy.special.ive

Notes
-----
We use the algorithm published by Clenshaw [1]_, and referenced by
Abramowitz and Stegun [2]_, for which the function domain is
partitioned into the two intervals [0,8] and (8,inf), and Chebyshev
polynomial expansions are employed in each interval. Relative error on
the domain [0,30] using IEEE arithmetic is documented [3]_ as having a
peak of 5.8e-16 with an rms of 1.4e-16 (n = 30000).

References
-----
.. [1] C. W. Clenshaw, "Chebyshev series for mathematical functions", in
    *National Physical Laboratory Mathematical Tables*, vol. 5, London:
    Her Majesty's Stationery Office, 1962.
.. [2] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical
    Functions*, 10th printing, New York: Dover, 1964, pp. 379.
    http://www.math.sfu.ca/~cbm/aands/page\_379.htm
.. [3] http://kobesearch.cpan.org/htdocs/Math-Cephes/Math/Cephes.html

Examples
-----
>>> np.i0([0.])
[1.0)
>>> np.i0([0., 1. + 2j])
[ 1.00000000+0.j      ,  0.18785373+0.64616944j]

identity(n, dtype=None)
Return the identity array.

The identity array is a square array with ones on
the main diagonal.

Parameters
-----
n : int
    Number of rows (and columns) in `n` x `n` output.
dtype : data-type, optional
    Data-type of the output. Defaults to ``float``.

Returns
-----
out : ndarray
    `n` x `n` array with its main diagonal set to one,
    and all other elements 0.

Examples
-----
>>> np.identity(3)
[[ 1.,  0.,  0.],
 [ 0.,  1.,  0.],
 [ 0.,  0.,  1.]]]

imag(val)
Return the imaginary part of the elements of the array.

Parameters
-----
val : array_like
    Input array.

Returns
-----
out : ndarray
    Output array. If `val` is real, the type of `val` is used for the
    output. If `val` has complex elements, the returned type is float.

See Also
-----
real, angle, real_if_close

Examples
-----
>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.imag
[ 2.,  4.,  6.]
>>> a.imag = np.array([8, 10, 12])
>>> a
[ 1. +8.j,  3.+10.j,  5.+12.j]
```

```
in1d(ar1, ar2, assume_unique=False, invert=False)
    Test whether each element of a 1-D array is also present in a second array.

    Returns a boolean array the same length as `ar1` that is True
    where an element of `ar1` is in `ar2` and False otherwise.

    Parameters
    -----
    ar1 : (M,) array_like
        Input array.
    ar2 : array_like
        The values against which to test each value of `ar1`.
    assume_unique : bool, optional
        If True, the input arrays are both assumed to be unique, which
        can speed up the calculation. Default is False.
    invert : bool, optional
        If True, the values in the returned array are inverted (that is,
        False where an element of `ar1` is in `ar2` and True otherwise).
        Default is False. ``np.in1d(a, b, invert=True)`` is equivalent
        to (but is faster than) ``np.invert(np.in1d(a, b))``.

    .. versionadded:: 1.8.0

    Returns
    -----
    in1d : (M,) ndarray, bool
        The values `ar1[in1d]` are in `ar2`.

    See Also
    -----
    numpy.lib.arraysetops : Module with a number of other functions for
                           performing set operations on arrays.

    Notes
    -----
    `in1d` can be considered as an element-wise function version of the
    python keyword `in`, for 1-D sequences. ``in1d(a, b)`` is roughly
    equivalent to ``np.array([item in b for item in a])``.
    However, this idea fails if `ar2` is a set, or similar (non-sequence)
    container: As ``ar2`` is converted to an array, in those cases
    ``asarray(ar2)`` is an object array rather than the expected array of
    contained values.

    .. versionadded:: 1.4.0

    Examples
    -----
    >>> test = np.array([0, 1, 2, 5, 0])
    >>> states = [0, 2]
    >>> mask = np.in1d(test, states)
    >>> mask
    array([ True, False,  True, False,  True], dtype=bool)
    >>> test[mask]
    array([0, 2, 0])
    >>> mask = np.in1d(test, states, invert=True)
    >>> mask
    array([False,  True, False,  True, False], dtype=bool)
    >>> test[mask]
    array([1, 5])

indices(dimensions, dtype=<type 'int'>)
    Return an array representing the indices of a grid.

    Compute an array where the subarrays contain index values 0,1, ...
    varying only along the corresponding axis.

    Parameters
    -----
    dimensions : sequence of ints
        The shape of the grid.
    dtype : dtype, optional
        Data type of the result.

    Returns
    -----
    grid : ndarray
        The array of grid indices,
        ``grid.shape = (len(dimensions),) + tuple(dimensions)``.

    See Also
    -----
    mgrid, meshgrid

    Notes
    -----
    The output shape is obtained by prepending the number of dimensions
    in front of the tuple of dimensions, i.e. if `dimensions` is a tuple
    ``r0, ..., rN-1`` of length ``N``, the output shape is
    ``r0, ..., rN-1``.
```

The subarrays ``grid[k]`` contains the N-D array of indices along the ``k-th`` axis. Explicitly::

```
grid[k,i0,i1,...,iN-1] = ik
```

Examples

```
-----
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]          # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]          # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```

Note that it would be more straightforward in the above example to extract the required elements directly with ``x[:2, :3]``.

info(object=None, maxwidth=76, output=<open file '<stdout>', mode 'w', toplevel='numpy')
Get help information for a function, class, or module.

Parameters

```
-----
object : object or str, optional
    Input object or name to get information about. If object is a
    numpy object, its docstring is given. If it is a string, available
    modules are searched for matching objects. If None, information
    about `info` itself is returned.
maxwidth : int, optional
    Printing width.
output : file like object, optional
    File like object that the output is written to, default is
    ``stdout``. The object has to be opened in 'w' or 'a' mode.
toplevel : str, optional
    Start search at this level.
```

See Also

`source`, `lookfor`

Notes

When used interactively with an `object`, ``np.info(obj)`` is equivalent to ``help(obj)`` on the Python prompt or ``obj?`` on the IPython prompt.

Examples

```
-----
>>> np.info(np.polyval) # doctest: +SKIP
polyval(p, x)
    Evaluate the polynomial p at x.
    ...
```

When using a string for `object` it is possible to get multiple results.

```
>>> np.info('fft') # doctest: +SKIP
    *** Found in numpy ***
Core FFT routines
...
    *** Found in numpy.fft ***
fft(a, n=None, axis=-1)
...
    *** Repeat reference found in numpy.fft.fftpack ***
    *** Total of 3 references found. ***
```

inner(...)

`inner`(a, b)

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without `complex` conjugation), in higher dimensions a sum product over the last axes.

Parameters

```
-----
a, b : array_like
    If a and b are nonscalar, their last dimensions must match.
```

```
Returns
-----
out : ndarray
    `out.shape = a.shape[:-1] + b.shape[:-1]`


Raises
-----
ValueError
    If the last dimension of `a` and `b` has different size.

See Also
-----
tensordot : Sum products over arbitrary axes.
dot : Generalised matrix product, using second last dimension of `b`.
einsum : Einstein summation convention.

Notes
-----
For vectors (1-D arrays) it computes the ordinary inner-product::

    np.inner(a, b) = sum(a[:]*b[:])

More generally, if `ndim(a) = r > 0` and `ndim(b) = s > 0`::

    np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))

or explicitly::

    np.inner(a, b)[i0,...,ir-1,j0,...,js-1]
        = sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])

In addition `a` or `b` may be scalars, in which case::

    np.inner(a,b) = a*b

Examples
-----
Ordinary inner product for vectors:

>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2

A multidimensional example:

>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])

An example where `b` is a scalar:

>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])


insert(arr, obj, values, axis=None)
Insert values along the given axis before the given indices.

Parameters
-----
arr : array_like
    Input array.
obj : int, slice or sequence of ints
    Object that defines the index or indices before which `values` is
    inserted.

.. versionadded:: 1.8.0

    Support for multiple insertions when `obj` is a single scalar or a
    sequence with one element (similar to calling insert multiple
    times).

values : array_like
    Values to insert into `arr`. If the type of `values` is different
    from that of `arr`, `values` is converted to the type of `arr`.
    `values` should be shaped so that ``arr[...,obj,...] = values```
    is legal.

axis : int, optional
    Axis along which to insert `values`. If `axis` is None then `arr`
    is flattened first.

Returns
-----
out : ndarray
    A copy of `arr` with `values` inserted. Note that `insert`
    does not occur in-place: a new array is returned. If
```

```

`axis` is None, `out` is a flattened array.

See Also
-----
append : Append elements at the end of an array.
concatenate : Join a sequence of arrays along an existing axis.
delete : Delete elements from an array.

Notes
-----
Note that for higher dimensional inserts `obj=0` behaves very different
from `obj=[0]` just like `arr[:,0,:] = values` is different from
`arr[:,[0],:] = values`.

Examples
-----
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, 2, 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])

Difference between sequence and scalars:

>>> np.insert(a, [1], [[1],[2],[3]], axis=1)
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])
>>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
...                  np.insert(a, [1], [[1],[2],[3]], axis=1))
True

>>> b = a.flatten()
>>> b
array([1, 1, 2, 2, 3, 3])
>>> np.insert(b, [2, 2], [5, 6])
array([1, 1, 5, 6, 2, 2, 3, 3])

>>> np.insert(b, slice(2, 4), [5, 6])
array([1, 1, 5, 2, 6, 2, 3, 3])
>>> np.insert(b, [2, 2], [7.13, False]) # type casting
array([1, 1, 7, 0, 2, 2, 3, 3])

>>> x = np.arange(8).reshape(2, 4)
>>> idx = (1, 3)
>>> np.insert(x, idx, 999, axis=1)
array([[ 0, 999,   1,   2, 999,   3],
       [ 4, 999,   5,   6, 999,   7]])

int_asbuffer(...)

interp(x, xp, fp, left=None, right=None, period=None)
One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function
with given values at discrete data-points.

Parameters
-----
x : array_like
    The x-coordinates of the interpolated values.

xp : 1-D sequence of floats
    The x-coordinates of the data points, must be increasing if argument
    `period` is not specified. Otherwise, `xp` is internally sorted after
    normalizing the periodic boundaries with ``xp = xp % period``.

fp : 1-D sequence of floats
    The y-coordinates of the data points, same length as `xp`.

left : float, optional
    Value to return for `x < xp[0]`, default is `fp[0]`.

right : float, optional
    Value to return for `x > xp[-1]`, default is `fp[-1]`.

period : None or float, optional
    A period for the x-coordinates. This parameter allows the proper
    interpolation of angular x-coordinates. Parameters `left` and `right`
    are ignored if `period` is specified.

.. versionadded:: 1.10.0

```

Returns

y : [float](#) or [ndarray](#)
 The interpolated values, same shape as `x`.

Raises

[ValueError](#)
 If `xp` and `fp` have different length
 If `xp` or `fp` are not 1-D sequences
 If `period == 0`

Notes

Does not check that the x-coordinate sequence `xp` is increasing.
If `xp` is not increasing, the results are nonsense.
A simple check for increasing is::

```
np.all(np.diff(xp) > 0)
```

Examples

>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
[array](#)([3., 3., 2.5, 0.56, 0.])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0

Plot an interpolant to the sine function:

```
>>> x = np.linspace(0, 2*np.pi, 10)  
>>> y = np.sin(x)  
>>> xvals = np.linspace(0, 2*np.pi, 50)  
>>> yinterp = np.interp(xvals, x, y)  
>>> import matplotlib.pyplot as plt  
>>> plt.plot(x, y, 'o')  
[<matplotlib.lines.Line2D object at 0x...>]  
>>> plt.plot(xvals, yinterp, '-x')  
[<matplotlib.lines.Line2D object at 0x...>]  
>>> plt.show()
```

Interpolation with periodic x-coordinates:

```
>>> x = [-180, -170, -185, 185, -10, -5, 0, 365]  
>>> xp = [190, -190, 350, -350]  
>>> fp = [5, 10, 3, 4]  
>>> np.interp(x, xp, fp, period=360)  
array([7.5, 5., 8.75, 6.25, 3., 3.25, 3.5, 3.75])
```

intersect1d(ar1, ar2, assume_unique=False)
Find the intersection of two arrays.

Return the sorted, unique values that are in both of the input arrays.

Parameters

ar1, ar2 : array_like
 Input arrays.
assume_unique : bool
 If True, the input arrays are both assumed to be unique, which
 can speed up the calculation. Default is False.

Returns

intersect1d : [ndarray](#)
 Sorted 1D array of common and unique elements.

See Also

numpy.lib.arraysetops : Module with a [number](#) of other functions for
 performing set operations on arrays.

Examples

>>> np.intersect1d([1, 3, 4, 3], [3, 1, 2, 1])
[array](#)([1, 3])

To intersect more than two arrays, use `functools.reduce`:

```
>>> from functools import reduce  
>>> reduce(np.intersect1d, ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2]))  
array([3])
```

```
ipmt(rate, per, nper, pv, fv=0.0, when='end')
    Compute the interest portion of a payment.

Parameters
-----
rate : scalar or array_like of shape(M, )
    Rate of interest as decimal (not per cent) per period
per : scalar or array_like of shape(M, )
    Interest paid against the loan changes during the life of the loan.
    The 'per' is the payment period to calculate the interest amount.
nper : scalar or array_like of shape(M, )
    Number of compounding periods
pv : scalar or array_like of shape(M, )
    Present value
fv : scalar or array_like of shape(M, ), optional
    Future value
when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
    When payments are due ('begin' (1) or 'end' (0)).
    Defaults to {'end', 0}.

Returns
-----
out : ndarray
    Interest portion of payment. If all input is scalar, returns a scalar
    float. If any input is array_like, returns interest payment for each
    input element. If multiple inputs are array_like, they all must have
    the same shape.

See Also
-----
ppmt, pmt, pv

Notes
-----
The total payment is made up of payment against principal plus interest.

```pmt = ppmt + ipmt``

Examples

What is the amortization schedule for a 1 year loan of $2500 at
8.24% interest per year compounded monthly?

>>> principal = 2500.00

The 'per' variable represents the periods of the loan. Remember that
financial equations start the period count at 1!

>>> per = np.arange(1*12) + 1
>>> ipmt = np.ipmt(0.0824/12, per, 1*12, principal)
>>> ppmt = np.ppmt(0.0824/12, per, 1*12, principal)

Each element of the sum of the 'ipmt' and 'ppmt' arrays should equal
'pmt'.

>>> pmt = np.pmt(0.0824/12, 1*12, principal)
>>> np.allclose(ipmt + ppmt, pmt)
True

>>> fmt = '{0:2d} {1:8.2f} {2:8.2f} {3:8.2f}'
>>> for payment in per:
... index = payment - 1
... principal = principal + ppmt[index]
... print(fmt.format(payment, ppmt[index], ipmt[index], principal))
1 -200.58 -17.17 2299.42
2 -201.96 -15.79 2097.46
3 -203.35 -14.40 1894.11
4 -204.74 -13.01 1689.37
5 -206.15 -11.60 1483.22
6 -207.56 -10.18 1275.66
7 -208.99 -8.76 1066.67
8 -210.42 -7.32 856.25
9 -211.87 -5.88 644.38
10 -213.32 -4.42 431.05
11 -214.79 -2.96 216.26
12 -216.26 -1.49 -0.00

>>> interestpd = np.sum(ipmt)
>>> np.round(interestpd, 2)
-112.98

irr(values)
 Return the Internal Rate of Return (IRR).

This is the "average" periodically compounded rate of return
that gives a net present value of 0.0; for a more complete explanation,
see Notes below.

Parameters
```

```

values : array_like, shape(N,)
 Input cash flows per time period. By convention, net "deposits"
 are negative and net "withdrawals" are positive. Thus, for
 example, at least the first element of `values`, which represents
 the initial investment, will typically be negative.

>Returns

out : float
 Internal Rate of Return for periodic input values.

The IRR is perhaps best understood through an example (illustrated
using np.irr in the Examples section below). Suppose one invests 100
units and then makes the following withdrawals at regular (fixed)
intervals: 39, 59, 55, 20. Assuming the ending value is 0, one's 100
unit investment yields 173 units; however, due to the combination of
compounding and the periodic withdrawals, the "average" rate of return
is neither simply 0.73/4 nor $(1.73)^{0.25}-1$. Rather, it is the solution
(for r) of the equation:

.. math:: -100 + \frac{39}{(1+r)} + \frac{59}{(1+r)^2} +
+ \frac{55}{(1+r)^3} + \frac{20}{(1+r)^4} = 0

In general, for `values` $= [v_0, v_1, \dots, v_M]$,
irr is the solution of the equation: [G]_

.. math:: \sum_{t=0}^M \frac{v_t}{(1+irr)^t} = 0


```

[References](#)

... [G] L. J. Gitman, "Principles of Managerial Finance, Brief," 3rd ed.,
Addison-Wesley, 2003, pg. 348.

[Examples](#)

```

>>> round(irr([-100, 39, 59, 55, 20]), 5)
0.28095
>>> round(irr([-100, 0, 0, 74]), 5)
-0.0955
>>> round(irr([-100, 100, 0, -7]), 5)
-0.0833
>>> round(irr([-100, 100, 0, 7]), 5)
0.06206
>>> round(irr([-5, 10.5, 1, -8, 1]), 5)
0.0886

```

(Compare with the Example given for `numpy.lib.financial.npv`)

**is\_busday(...)**

```

is_busday(dates, weekmask='1111100', holidays=None, busdaycal=None, out=None)

Calculates which of the given dates are valid days, and which are not.

.. versionadded:: 1.7.0

dates : array_like of datetime64\[D\]
 The array of dates to process.
weekmask : str or array_like of bool, optional
 A seven-element array indicating which of Monday through Sunday are
 valid days. May be specified as a length-seven list or array, like
 [1,1,1,1,0,0,0]; a length-seven string, like '1111100'; or a string
 like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for
 weekdays, optionally separated by white space. Valid abbreviations
 are: Mon Tue Wed Thu Fri Sat Sun
holidays : array_like of datetime64\[D\], optional
 An array of dates to consider as invalid dates. They may be
 specified in any order, and NaT (not-a-time) dates are ignored.
 This list is saved in a normalized form that is suited for
 fast calculations of valid days.
busdaycal : busdaycalendar, optional
 A `busdaycalendar` object which specifies the valid days. If this
 parameter is provided, neither weekmask nor holidays may be
 provided.
out : array of bool, optional
 If provided, this array is filled with the result.

>Returns

out : array of bool
 An array with the same shape as ``dates``, containing True for
 each valid day, and False for each invalid day.


```

[See Also](#)

**busdaycalendar**: An [object](#) that specifies a custom set of valid days.  
**busday\_offset** : Applies an offset counted in valid days.  
**busday\_count** : Counts how many valid days are in a [half](#)-open date range.

**Examples**

```
>>> # The weekdays are Friday, Saturday, and Monday
... np.is_busday(['2011-07-01', '2011-07-02', '2011-07-18'],
... holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
array([False, False, True], dtype='bool')
```

**isclose(a, b, rtol=1e-05, atol=1e-08, equal\_nan=False)**  
 Returns a boolean array where two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (`'rtol' * abs('b')`) and the absolute difference `'atol'` are added together to compare against the absolute difference between `'a'` and `'b'`.

**Parameters**

```
a, b : array_like
 Input arrays to compare.
rtol : float
 The relative tolerance parameter (see Notes).
atol : float
 The absolute tolerance parameter (see Notes).
equal_nan : bool
 Whether to compare NaN's as equal. If True, NaN's in 'a' will be
 considered equal to NaN's in 'b' in the output array.
```

**Returns**

```
y : array_like
 Returns a boolean array of where 'a' and 'b' are equal within the given tolerance. If both 'a' and 'b' are scalars, returns a single boolean value.
```

**See Also**

```
allclose
```

**Notes**

```
.. versionadded:: 1.7.0
```

For finite values, `isclose` uses the following equation to test whether two [floating](#) point values are equivalent.

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

The above equation is not symmetric in `'a'` and `'b'`, so that `'isclose(a, b)'` might be different from `'isclose(b, a)'` in some rare cases.

**Examples**

```
>>> np.isclose([1e10,1e-7], [1.00001e10,1e-8])
array([True, False])
>>> np.isclose([1e10,1e-8], [1.00001e10,1e-9])
array([True, True])
>>> np.isclose([1e10,1e-8], [1.0001e10,1e-9])
array([False, True])
>>> np.isclose([1.0, np.nan], [1.0, np.nan])
array([True, False])
>>> np.isclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
array([True, True])
```

**iscomplex(x)**  
 Returns a bool array, where True if input element is [complex](#).

What is tested is whether the input has a non-zero imaginary part, not if the input type is [complex](#).

**Parameters**

```
x : array_like
 Input array.
```

**Returns**

```
out : ndarray of bools
 Output array.
```

**See Also**

```
isreal
```

iscomplexobj : Return True if x is a [complex](#) type or an array of [complex](#) numbers.

Examples

-----

```
>>> np.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([True, False, False, False, False, True], dtype=bool)
```

**iscomplexobj(x)**

Check for a [complex](#) type or an array of [complex](#) numbers.

The type of the input is checked, not the value. Even if the input has an imaginary part equal to zero, `iscomplexobj` evaluates to True.

Parameters

-----

x : any  
The input can be of any type and shape.

Returns

-----

iscomplexobj : bool  
The return value, True if 'x' is of a [complex](#) type or has at least one [complex](#) element.

See Also

-----

[isrealobj](#), [iscomplex](#)

Examples

-----

```
>>> np.iscomplexobj(1)
False
>>> np.iscomplexobj(1+0j)
True
>>> np.iscomplexobj([3, 1+0j, True])
True
```

**isfortran(a)**

Returns True if the array is Fortran contiguous but \*not\* C contiguous.

This function is obsolete and, because of changes due to relaxed stride checking, its return value for the same array may differ for versions of Numpy >= 1.10 and previous versions. If you only want to check if an array is Fortran contiguous use ``a.flags.f\_contiguous`` instead.

Parameters

-----

a : [ndarray](#)  
Input array.

Examples

-----

np.array allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
 [4, 5, 6]])
>>> np.isfortran(a)
False

>>> b = np.array([[1, 2, 3], [4, 5, 6]], order='FORTRAN')
>>> b
array([[1, 2, 3],
 [4, 5, 6]])
>>> np.isfortran(b)
True
```

The transpose of a C-ordered array is a FORTRAN-ordered array.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
 [4, 5, 6]])
>>> np.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
 [2, 5],
 [3, 6]])
>>> np.isfortran(b)
```

```
True
C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.
>>> np.isfortran(np.array([1, 2], order='FORTRAN'))
False

isneginf(x, y=None)
Test element-wise for negative infinity, return result as bool array.

Parameters

x : array_like
 The input array.
y : array_like, optional
 A boolean array with the same shape and type as `x` to store the
 result.

Returns

y : ndarray
 A boolean array with the same dimensions as the input.
 If second argument is not supplied then a numpy boolean array is
 returned with values True where the corresponding element of the
 input is negative infinity and values False where the element of
 the input is not negative infinity.

 If a second argument is supplied the result is stored there. If the
 type of that array is a numeric type the result is represented as
 zeros and ones, if the type is boolean then as False and True. The
 return value `y` is then a reference to that array.

See Also

isinf, isposinf, isnan, isfinite

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic
(IEEE 754).

Errors result if the second argument is also supplied when x is a scalar
input, or if first and second arguments have different shapes.

Examples

>>> np.isneginf(np.NINF)
array(True, dtype=bool)
>>> np.isneginf(np.inf)
array(False, dtype=bool)
>>> np.isneginf(np.PINF)
array(False, dtype=bool)
>>> np.isneginf([-np.inf, 0., np.inf])
array([True, False, False], dtype=bool)

>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isneginf(x, y)
array([1, 0, 0])
>>> y
array([1, 0, 0])

isposinf(x, y=None)
Test element-wise for positive infinity, return result as bool array.

Parameters

x : array_like
 The input array.
y : array_like, optional
 A boolean array with the same shape as `x` to store the result.

Returns

y : ndarray
 A boolean array with the same dimensions as the input.
 If second argument is not supplied then a boolean array is returned
 with values True where the corresponding element of the input is
 positive infinity and values False where the element of the input is
 not positive infinity.

 If a second argument is supplied the result is stored there. If the
 type of that array is a numeric type the result is represented as zeros
 and ones, if the type is boolean then as False and True.
 The return value `y` is then a reference to that array.

See Also

```

```
isinf, isneginf, isfinite, isnan

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
(IEEE 754).

Errors result if the second argument is also supplied when `x` is a
scalar input, or if first and second arguments have different shapes.

Examples

>>> np.isposinf(np.PINF)
array(True, dtype=bool)
>>> np.isposinf(np.inf)
array(True, dtype=bool)
>>> np.isposinf(np.NINF)
array(False, dtype=bool)
>>> np.isposinf([-np.inf, 0., np.inf])
array([False, False, True], dtype=bool)

>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isposinf(x, y)
array([0, 0, 1])
>>> y
array([0, 0, 1])
```

**isreal(x)**

Returns a bool array, where True if input element is real.

If element has [complex](#) type with zero [complex](#) part, the return value
for that element is True.

**Parameters**

x : array\_like  
Input array.

**Returns**

out : [ndarray](#), bool  
Boolean array of same shape as `x`.

**See Also**

[iscomplex](#)  
[isrealobj](#) : Return True if x is not a [complex](#) type.

**Examples**

```
>>> np.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([False, True, True, True, False], dtype=bool)
```

**isrealobj(x)**

Return True if x is a not [complex](#) type or an array of [complex](#) numbers.

The type of the input is checked, not the value. So even if the input
has an imaginary part equal to zero, `isrealobj` evaluates to False
if the data type is [complex](#).

**Parameters**

x : any  
The input can be of any type and shape.

**Returns**

y : bool  
The return value, False if `x` is of a [complex](#) type.

**See Also**

[iscomplexobj](#), [isreal](#)

**Examples**

```
>>> np.isrealobj(1)
True
>>> np.isrealobj(1+0j)
False
>>> np.isrealobj([3, 1+0j, True])
False
```

**isscalar(num)**

Returns True if the type of `num` is a scalar type.

**Parameters**

```

num : any
 Input argument, can be of any type and shape.

Returns

val : bool
 True if `num` is a scalar type, False if it is not.

Examples

>>> np.isscalar(3.1)
True
>>> np.isscalar([3.1])
False
>>> np.isscalar(False)
True

issctype(rep)
Determine whether the given object represents a scalar data-type.

Parameters

rep : any
 If `rep` is an instance of a scalar dtype, True is returned. If not,
 False is returned.

Returns

out : bool
 Boolean result of check whether `rep` is a scalar dtype.

See Also

issubctype, issubdtype, obj2sctype, sctype2char

Examples

>>> np.issctype(np.int32)
True
>>> np.issctype(list)
False
>>> np.issctype(1.1)
False

Strings are also a scalar type:
>>> np.issctype(np.dtype('str'))
True

issubclass_(arg1, arg2)
Determine if a class is a subclass of a second class.

`issubclass_` is equivalent to the Python built-in ``issubclass```,
except that it returns False instead of raising a TypeError if one
of the arguments is not a class.

Parameters

arg1 : class
 Input class. True is returned if `arg1` is a subclass of `arg2`.
arg2 : class or tuple of classes.
 Input class. If a tuple of classes, True is returned if `arg1` is a
 subclass of any of the tuple elements.

Returns

out : bool
 Whether `arg1` is a subclass of `arg2` or not.

See Also

issubctype, issubdtype, issctype

Examples

>>> np.issubclass_(np.int32, np.int)
True
>>> np.issubclass_(np.int32, np.float)
False

issubdtype(arg1, arg2)
Returns True if first argument is a typecode lower/equal in type hierarchy.

Parameters

arg1, arg2 : dtype_like
 dtype or string representing a typecode.
```

```
Returns

out : bool

See Also

issubctype, issubclass_
numpy.core.numericatypes : Overview of numpy type hierarchy.

Examples

>>> np.issubdtype('S1', str)
True
>>> np.issubdtype(np.float64, np.float32)
False

issubsctype(arg1, arg2)
Determine if the first argument is a subclass of the second argument.

Parameters

arg1, arg2 : dtype or dtype specifier
 Data-types.

Returns

out : bool
 The result.

See Also

issctype, issubdtype,obj2sctype

Examples

>>> np.issubsctype('S8', str)
True
>>> np.issubsctype(np.array([1]), np.int)
True
>>> np.issubsctype(np.array([1]), np.float)
False

iterable(y)
Check whether or not an object can be iterated over.

Parameters

y : object
 Input object.

Returns

b : {0, 1}
 Return 1 if the object has an iterator method or is a sequence,
 and 0 otherwise.

Examples

>>> np.iterable([1, 2, 3])
1
>>> np.iterable(2)
0

ix_(*args)
Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N
dimensions each, such that the shape is 1 in all but one dimension
and the dimension with the non-unit shape value cycles through all
N dimensions.

Using `ix_` one can quickly construct index arrays that will index
the cross product. ``a[np.ix_([1,3],[2,5])`` returns the array
``[[a[1,2] a[1,5]], [a[3,2] a[3,5]]``.

Parameters

args : 1-D sequences

Returns

out : tuple of ndarrays
 N arrays with N dimensions each, with N the number of input
 sequences. Together these arrays form an open mesh.

See Also
```

```

ogrid, mgrid, meshgrid

Examples

>>> a = np.arange(10).reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]])
>>> ixgrid = np.ix_([0,1], [2,4])
>>> ixgrid
(array([0,
 1]), array([[2, 4]]))
>>> ixgrid[0].shape, ixgrid[1].shape
((2, 1), (1, 2))
>>> a[ixgrid]
array([[2, 4],
 [7, 9]])

kaiser(M, beta)
Return the Kaiser window.

The Kaiser window is a taper formed by using a Bessel function.

Parameters

M : int
 Number of points in the output window. If zero or less, an
 empty array is returned.
beta : float
 Shape parameter for window.

Returns

out : array
 The window, with the maximum value normalized to one (the value
 one appears only if the number of samples is odd).

See Also

bartlett, blackman, hamming, hanning

Notes

The Kaiser window is defined as

.. math:: w(n) = I_0\left(\beta \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta)

with

.. math:: \quad -\frac{M-1}{2} \leq n \leq \frac{M-1}{2},

where :math:`I_0` is the modified zeroth-order Bessel function.

The Kaiser was named for Jim Kaiser, who discovered a simple
approximation to the DPSS window based on Bessel functions. The Kaiser
window is a very good approximation to the Digital Prolate Spheroidal
Sequence, or Slepian window, which is the transform which maximizes the
energy in the main lobe of the window relative to total energy.

The Kaiser can approximate many other windows by varying the beta
parameter.

=====
beta Window shape
=====
0 Rectangular
5 Similar to a Hamming
6 Similar to a Hanning
8.6 Similar to a Blackman
=====

A beta value of 14 is probably a good starting point. Note that as beta
gets large, the window narrows, and so the number of samples needs to be
large enough to sample the increasingly narrow spike, otherwise NaNs will
get returned.

Most references to the Kaiser window come from the signal processing
literature, where it is used as one of many windowing functions for
smoothing values. It is also known as an apodization (which means
"removing the foot", i.e. smoothing discontinuities at the beginning
and end of the sampled signal) or tapering function.

References

.. [1] J. F. Kaiser, "Digital Filters" - Ch 7 in "Systems analysis by
 digital computer", Editors: F.F. Kuo and J.F. Kaiser, p 218-285.
 John Wiley and Sons, New York, (1966).

```

.. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 177-178.  
 .. [3] Wikipedia, "Window function",  
[http://en.wikipedia.org/wiki/Window\\_function](http://en.wikipedia.org/wiki/Window_function)

**Examples**

```

>>> np.kaiser(12, 14)
array([7.72686684e-06, 3.46009194e-03, 4.65200189e-02,
 2.29737120e-01, 5.99885316e-01, 9.45674898e-01,
 9.45674898e-01, 5.99885316e-01, 2.29737120e-01,
 4.65200189e-02, 3.46009194e-03, 7.72686684e-06])
```

Plot the window and the frequency response:

```
>>> from numpy.fft import fft, fftshift
>>> window = np.kaiser(51, 14)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Kaiser window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Kaiser window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()
```

**kron(a, b)**

Kronecker product of two arrays.

Computes the Kronecker product, a composite array made of blocks of the second array scaled by the first.

**Parameters**

-----  
`a, b : array_like`

**Returns**

-----  
`out : ndarray`

**See Also**

-----  
`outer : The outer product`

**Notes**

-----  
 The function assumes that the `number` of dimensions of `a` and `b` are the same, if necessary prepending the smallest with ones.  
 If `a.shape = (r0,r1,...,rN)` and `b.shape = (s0,s1,...,sN)`, the Kronecker product has shape `(r0*s0, r1*s1, ..., rN*sN)`.  
 The elements are products of elements from `a` and `b`, organized explicitly by::

`kron(a,b)[k0,k1,...,kN] = a[i0,i1,...,iN] * b[j0,j1,...,jN]`

where::

`kt = it * st + jt, t = 0,...,N`

In the common 2-D case (N=1), the block structure can be visualized::

```
[[a[0,0]*b, a[0,1]*b, ... , a[0,-1]*b],
 [... ...],
 [a[-1,0]*b, a[-1,1]*b, ... , a[-1,-1]*b]]
```

**Examples**

```

>>> np.kron([1,10,100], [5,6,7])
[5, 6, 7, 50, 60, 70, 500, 600, 700]
>>> np.kron([5,6,7], [1,10,100])
[5, 50, 500, 6, 60, 600, 7, 70, 700]

>>> np.kron(np.eye(2), np.ones((2,2)))
[[1., 1., 0., 0.],
 [1., 1., 0., 0.],
 [0., 0., 1., 1.],
 [0., 0., 1., 1.]]

>>> a = np.arange(100).reshape((2,5,2,5))
>>> b = np.arange(24).reshape((2,3,4))
>>> c = np.kron(a,b)
>>> c.shape
(2, 10, 6, 20)
>>> I = (1,3,0,2)
>>> J = (0,2,1)
>>> J1 = (0,) + J # extend to ndim=4
>>> S1 = (1,) + b.shape
>>> K = tuple(np.array(I) * np.array(S1) + np.array(J1))
>>> c[K] == a[I]*b[J]
True

lexsort(...)
lexsort(keys, axis=-1)

Perform an indirect sort using a sequence of keys.

Given multiple sorting keys, which can be interpreted as columns in a
spreadsheet, lexsort returns an array of integer indices that describes
the sort order by multiple columns. The last key in the sequence is used
for the primary sort order, the second-to-last key for the secondary sort
order, and so on. The keys argument must be a sequence of objects that
can be converted to arrays of the same shape. If a 2D array is provided
for the keys argument, it's rows are interpreted as the sorting keys and
sorting is according to the last row, second last row etc.

Parameters

keys : (k, N) array or tuple containing k (N,)-shaped sequences
 The `k` different "columns" to be sorted. The last column (or row if
 `keys` is a 2D array) is the primary sort key.
axis : int, optional
 Axis to be indirectly sorted. By default, sort over the last axis.

Returns

indices : (N,) ndarray of ints
 Array of indices that sort the keys along the specified axis.

See Also

argsort : Indirect sort.
ndarray.sort : In-place sort.
sort : Return a sorted copy of an array.

Examples

Sort names: first by surname, then by name.

>>> surnames = ('Hertz', 'Galilei', 'Hertz')
>>> first_names = ('Heinrich', 'Galileo', 'Gustav')
>>> ind = np.lexsort((first_names, surnames))
>>> ind
[1, 2, 0]

>>> [surnames[i] + ", " + first_names[i] for i in ind]
['Galilei', 'Galileo', 'Hertz', 'Gustav', 'Hertz', 'Heinrich']

Sort two columns of numbers:

>>> a = [1,5,1,4,3,4,4] # First column
>>> b = [9,4,0,4,0,2,1] # Second column
>>> ind = np.lexsort((b,a)) # Sort by a, then by b
>>> print(ind)
[2 0 4 6 5 3 1]

>>> [(a[i],b[i]) for i in ind]
[(1, 0), (1, 9), (3, 0), (4, 1), (4, 2), (4, 4), (5, 4)]

Note that sorting is first according to the elements of ``a``.
Secondary sorting is according to the elements of ``b``.

A normal ``argsort`` would have yielded:

>>> [(a[i],b[i]) for i in np.argsort(a)]
[(1, 0), (1, 0), (3, 0), (4, 4), (4, 2), (4, 1), (5, 4)]

```

```

Structured arrays are sorted lexically by ``argsort``:
>>> x = np.array([(1,9), (5,4), (1,0), (4,4), (3,0), (4,2), (4,1)],
... dtype=np.dtype([('x', int), ('y', int)]))
>>> np.argsort(x) # or np.argsort(x, order=('x', 'y'))
array([2, 0, 4, 6, 5, 3, 1])

linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

Parameters

start : scalar
 The starting value of the sequence.
stop : scalar
 The end value of the sequence, unless `endpoint` is set to False.
 In that case, the sequence consists of all but the last of ``num + 1``
 evenly spaced samples, so that `stop` is excluded. Note that the step
 size changes when `endpoint` is False.
num : int, optional
 Number of samples to generate. Default is 50. Must be non-negative.
endpoint : bool, optional
 If True, `stop` is the last sample. Otherwise, it is not included.
 Default is True.
retstep : bool, optional
 If True, return (`samples`, `step`), where `step` is the spacing
 between samples.
dtype : dtype, optional
 The type of the output array. If `dtype` is not given, infer the data
 type from the other input arguments.

.. versionadded:: 1.9.0

Returns

samples : ndarray
 There are `num` equally spaced samples in the closed interval
 ``[start, stop]`` or the half-open interval ``[start, stop)``
 (depending on whether `endpoint` is True or False).
step : float
 Only returned if `retstep` is True

 Size of spacing between samples.

See Also

arange : Similar to `linspace`, but uses a step size (instead of the
 number of samples).
logspace : Samples uniformly distributed in log space.

Examples

>>> np.linspace(2.0, 3.0, num=5)
array([2. , 2.25, 2.5, 2.75, 3.])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2. , 2.25, 2.5, 2.75, 3.]), 0.25)

Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()

load(file, mmap_mode=None, allow_pickle=True, fix_imports=True, encoding='ASCII')
Load arrays or pickled objects from ``.npy``, ``.npz`` or pickled files.

Parameters

file : file-like object or string
 The file to read. File-like objects must support the

```

```

``seek()`` and ``read()`` methods. Pickled files require that the
file-like object support the ``readline()`` method as well.
mmap_mode : {None, 'r+', 'r', 'w+', 'c'}, optional
 If not None, then memory-map the file, using the given mode (see
 `numpy.memmap` for a detailed description of the modes). A
 memory-mapped array is kept on disk. However, it can be accessed
 and sliced like any ndarray. Memory mapping is especially useful
 for accessing small fragments of large files without reading the
 entire file into memory.
allow_pickle : bool, optional
 Allow loading pickled object arrays stored in npy files. Reasons for
 disallowing pickles include security, as loading pickled data can
 execute arbitrary code. If pickles are disallowed, loading object
 arrays will fail.
 Default: True
fix_imports : bool, optional
 Only useful when loading Python 2 generated pickled files on Python 3,
 which includes npy/npz files containing object arrays. If `fix_imports`
 is True, pickle will try to map the old Python 2 names to the new names
 used in Python 3.
encoding : str, optional
 What encoding to use when reading Python 2 strings. Only useful when
 loading Python 2 generated pickled files on Python 3, which includes
 npy/npz files containing object arrays. Values other than 'latin1',
 'ASCII', and 'bytes' are not allowed, as they can corrupt numerical
 data. Default: 'ASCII'

Returns

result : array, tuple, dict, etc.
 Data stored in the file. For ```.npz``` files, the returned instance
 of NpzFile class must be closed to avoid leaking file descriptors.

Raises

IOError
 If the input file does not exist or cannot be read.
ValueError
 The file contains an object array, but allow_pickle=False given.

See Also

save, savez, savez_compressed, loadtxt
memmap : Create a memory-map to an array stored in a file on disk.

Notes

- If the file contains pickle data, then whatever object is stored
 in the pickle is returned.
- If the file is a ```.npy``` file, then a single array is returned.
- If the file is a ```.npz``` file, then a dictionary-like object is
 returned, containing ``{filename: array}`` key-value pairs, one for
 each file in the archive.
- If the file is a ```.npz``` file, the returned value supports the
 context manager protocol in a similar fashion to the open function::

 with load('foo.npz') as data:
 a = data['a']

 The underlying file descriptor is closed when exiting the 'with'
 block.

Examples

Store data to disk, and load it again:

>>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
>>> np.load('/tmp/123.npy')
array([[1, 2, 3],
 [4, 5, 6]])

Store compressed data to disk, and load it again:

>>> a=np.array([[1, 2, 3], [4, 5, 6]])
>>> b=np.array([1, 2])
>>> np.savez('/tmp/123.npz', a=a, b=b)
>>> data = np.load('/tmp/123.npz')
>>> data['a']
array([[1, 2, 3],
 [4, 5, 6]])
>>> data['b']
array([1, 2])
>>> data.close()

Mem-map the stored array, and then access the second row
directly from disk:

>>> X = np.load('/tmp/123.npy', mmap_mode='r')
>>> X[1, :]

```

```
memmap([4, 5, 6])

loads(...)

 loads(string) -- Load a pickle from the given string

loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0,
usecols=None, unpack=False, ndmin=0)
 Load data from a text file.

 Each row in the text file must have the same number of values.

 Parameters

 fname : file or str
 File, filename, or generator to read. If the filename extension is
 ``.gz`` or ``.bz2``, the file is first decompressed. Note that
 generators should return byte strings for Python 3k.
 dtype : data-type, optional
 Data-type of the resulting array; default: float. If this is a
 structured data-type, the resulting array will be 1-dimensional, and
 each row will be interpreted as an element of the array. In this
 case, the number of columns used must match the number of fields in
 the data-type.
 comments : str or sequence, optional
 The characters or list of characters used to indicate the start of a
 comment;
 default: '#'.
 delimiter : str, optional
 The string used to separate values. By default, this is any
 whitespace.
 converters : dict, optional
 A dictionary mapping column number to a function that will convert
 that column to a float. E.g., if column 0 is a date string:
 ``converters = {0: datestr2num}``. Converters can also be used to
 provide a default value for missing data (but see also `genfromtxt`):
 ``converters = {3: lambda s: float(s.strip() or 0)}``. Default: None.
 skiprows : int, optional
 Skip the first `skiprows` lines; default: 0.
 usecols : sequence, optional
 Which columns to read, with 0 being the first. For example,
 ``usecols = (1,4,5)`` will extract the 2nd, 5th and 6th columns.
 The default, None, results in all columns being read.
 unpack : bool, optional
 If True, the returned array is transposed, so that arguments may be
 unpacked using ``x, y, z = loadtxt(...)``. When used with a structured
 data-type, arrays are returned for each field. Default is False.
 ndmin : int, optional
 The returned array will have at least `ndmin` dimensions.
 Otherwise mono-dimensional axes will be squeezed.
 Legal values: 0 (default), 1 or 2.

 .. versionadded:: 1.6.0

 Returns

 out : ndarray
 Data read from the text file.

 See Also

 load, fromstring, fromregex
 genfromtxt : Load data with missing values handled as specified.
 scipy.io.loadmat : reads MATLAB data files

 Notes

 This function aims to be a fast reader for simply formatted files. The
 `genfromtxt` function provides more sophisticated handling of, e.g.,
 lines with missing values.

 .. versionadded:: 1.10.0

 The strings produced by the Python float.hex method can be used as
 input for floats.

 Examples

 >>> from io import StringIO # StringIO behaves like a file object
 >>> c = StringIO("0 1\n2 3")
 >>> np.loadtxt(c)
 array([[0., 1.],
 [2., 3.]])

 >>> d = StringIO("M 21 72\nF 35 58")
 >>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
 ... 'formats': ('S1', 'i4', 'f4')})
 array([('M', 21, 72.0), ('F', 35, 58.0)],
 dtype=[('gender', '|S1'), ('age', '
```

```

>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([1., 3.])
>>> y
array([2., 4.])

logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)
 Return numbers spaced evenly on a log scale.

 In linear space, the sequence starts at ``base ** start`` ('base' to the power of 'start') and ends with ``base ** stop`` (see 'endpoint' below).

 Parameters

 start : float
 ``base ** start`` is the starting value of the sequence.
 stop : float
 ``base ** stop`` is the final value of the sequence, unless `endpoint` is False. In that case, ``num + 1`` values are spaced over the interval in log-space, of which all but the last (a sequence of length ``num``) are returned.
 num : integer, optional
 Number of samples to generate. Default is 50.
 endpoint : boolean, optional
 If true, 'stop' is the last sample. Otherwise, it is not included.
 Default is True.
 base : float, optional
 The base of the log space. The step size between the elements in ``ln(samples) / ln(base)`` (or ``log_base(samples)``) is uniform.
 Default is 10.0.
 dtype : dtype
 The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

 Returns

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

 See Also

 arange : Similar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.
 linspace : Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.

 Notes

 Logspace is equivalent to the code

 >>> y = np.linspace(start, stop, num=num, endpoint=endpoint)
 ... # doctest: +SKIP
 >>> power(base, y).astype(dtype)
 ... # doctest: +SKIP

 Examples

 >>> np.logspace(2.0, 3.0, num=4)
 array([100. , 215.443469 , 464.15888336, 1000.])
 >>> np.logspace(2.0, 3.0, num=4, endpoint=False)
 array([100. , 177.827941 , 316.22776602, 562.34132519])
 >>> np.logspace(2.0, 3.0, num=4, base=2.0)
 array([4. , 5.0396842 , 6.34960421, 8.])

 Graphical illustration:

 >>> import matplotlib.pyplot as plt
 >>> N = 10
 >>> x1 = np.logspace(0.1, 1, N, endpoint=True)
 >>> x2 = np.logspace(0.1, 1, N, endpoint=False)
 >>> y = np.zeros(N)
 >>> plt.plot(x1, y, 'o')
 [<matplotlib.lines.Line2D object at 0x...>]
 >>> plt.plot(x2, y + 0.5, 'o')
 [<matplotlib.lines.Line2D object at 0x...>]
 >>> plt.ylim([-0.5, 1])
 (-0.5, 1)
 >>> plt.show()

lookfor(what, module=None, import_modules=True, regenerate=False, output=None)
 Do a keyword search on docstrings.

 A list of objects that matched the search is displayed,
 sorted by relevance. All given keywords need to be found in the

```

```
docstring for it to be returned as a result, but the order does
not matter.

Parameters

what : str
 String containing words to look for.
module : str or list, optional
 Name of module(s) whose docstrings to go through.
import_modules : bool, optional
 Whether to import sub-modules in packages. Default is True.
regenerate : bool, optional
 Whether to re-generate the docstring cache. Default is False.
output : file-like, optional
 File-like object to write the output to. If omitted, use a pager.

See Also

source, info

Notes

Relevance is determined only roughly, by checking if the keywords occur
in the function name, at the start of a docstring, etc.

Examples

>>> np.lookfor('binary representation')
Search results for 'binary representation'

numpy.binary_repr
 Return the binary representation of the input number as a string.
numpy.core.setup_common.long_double_representation
 Given a binary dump as given by GNU od -b, look for long double
numpy.base_repr
 Return a string representation of a number in the given base system.
...
mafromtxt(fname, **kwargs)
Load ASCII data stored in a text file and return a masked array.

Parameters

fname, kwargs : For a description of input parameters, see `genfromtxt`.

See Also

numpy.genfromtxt : generic function to load ASCII data.

mask_indices(n, mask_func, k=0)
Return the indices to access (n, n) arrays, given a masking function.

Assume `mask_func` is a function that, for a square array a of size
``(n, n)`` with a possible offset argument `k`, when called as
``mask_func(a, k)`` returns a new array with zeros in certain locations
(functions like `triu` or `tril` do precisely this). Then this function
returns the indices where the non-zero values would be located.

Parameters

n : int
 The returned indices will be valid to access arrays of shape (n, n).
mask_func : callable
 A function whose call signature is similar to that of `triu`, `tril`.
 That is, ``mask_func(x, k)`` returns a boolean array, shaped like `x`.
 `k` is an optional argument to the function.
k : scalar
 An optional argument which is passed through to `mask_func`. Functions
 like `triu`, `tril` take a second argument that is interpreted as an
 offset.

Returns

indices : tuple of arrays.
 The `n` arrays of indices corresponding to the locations where
 ``mask_func(np.ones((n, n)), k)`` is True.

See Also

triu, tril, triu_indices, tril_indices

Notes

.. versionadded:: 1.4.0

Examples

These are the indices that would allow you to access the upper triangular
```

```
part of any 3x3 array:
>>> iu = np.mask_indices(3, np.triu)
For example, if `a` is a 3x3 array:
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
 [3, 4, 5],
 [6, 7, 8]])
>>> a[iu]
array([0, 1, 2, 4, 5, 8])
An offset can be passed also to the masking function. This gets us the
indices starting on the first diagonal right of the main one:
>>> iu1 = np.mask_indices(3, np.triu, 1)
with which we now extract only three elements:
>>> a[iu1]
array([1, 2, 5])
mat = asmatrix(data, dtype=None)
Interpret the input as a matrix.
Unlike 'matrix', `asmatrix` does not make a copy if the input is already
a matrix or an ndarray. Equivalent to ``matrix(data, copy=False)``.
Parameters

data : array_like
 Input data.
dtype : data-type
 Data-type of the output matrix.
Returns

mat : matrix
 'data' interpreted as a matrix.
Examples

>>> x = np.array([[1, 2], [3, 4]])
>>> m = np.asmatrix(x)
>>> x[0,0] = 5
>>> m
matrix([[5, 2],
 [3, 4]])
matmul(...)
matmul(a, b, out=None)
Matrix product of two arrays.
The behavior depends on the arguments in the following way.
- If both arguments are 2-D they are multiplied like conventional
 matrices.
- If either argument is N-D, N > 2, it is treated as a stack of
 matrices residing in the last two indexes and broadcast accordingly.
- If the first argument is 1-D, it is promoted to a matrix by
 prepending a 1 to its dimensions. After matrix multiplication
 the prepended 1 is removed.
- If the second argument is 1-D, it is promoted to a matrix by
 appending a 1 to its dimensions. After matrix multiplication
 the appended 1 is removed.
Multiplication by a scalar is not allowed, use ``*`` instead. Note that
multiplying a stack of matrices with a vector will result in a stack of
vectors, but matmul will not recognize it as such.
``matmul`` differs from ``dot`` in two important ways.
- Multiplication by scalars is not allowed.
- Stacks of matrices are broadcast together as if the matrices
 were elements.
.. warning::
 This function is preliminary and included in Numpy 1.10 for testing
 and documentation. Its semantics will not change, but the number and
 order of the optional arguments will.
.. versionadded:: 1.10.0
```

**Parameters**  
-----  
**a** : array\_like  
First argument.  
**b** : array\_like  
Second argument.  
**out** : [ndarray](#), optional  
Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its [dtype](#) must be the [dtype](#) that would be returned for `'dot'(a,b)`. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be [flexible](#).

**Returns**  
-----  
**output** : [ndarray](#)  
Returns the dot product of `a` and `b`. If `a` and `b` are both 1-D arrays then a scalar is returned; otherwise an array is returned. If `out` is given, then it is returned.

**Raises**  
-----  
**ValueError**  
If the last dimension of `a` is not the same size as the second-to-last dimension of `b`.  
If scalar value is passed.

**See Also**  
-----  
**vdot** : Complex-conjugating dot product.  
**tensordot** : Sum products over arbitrary axes.  
**einsum** : Einstein summation convention.  
**dot** : alternative [matrix](#) product with different broadcasting rules.

**Notes**  
-----  
The `matmul` function implements the semantics of the `@` operator introduced in Python 3.5 following [PEP465](#).

**Examples**  
-----  
For 2-D arrays it is the [matrix](#) product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.matmul(a, b)
array([[4, 1],
 [2, 2]])
```

For 2-D mixed with 1-D, the result is the usual.

```
>>> a = [[1, 0], [0, 1]]
>>> b = [1, 2]
>>> np.matmul(a, b)
array([1, 2])
>>> np.matmul(b, a)
array([1, 2])
```

Broadcasting is conventional for stacks of arrays

```
>>> a = np.arange(2*2*4).reshape((2,2,4))
>>> b = np.arange(2*2*4).reshape((2,4,2))
>>> np.matmul(a,b).shape
(2, 2, 2)
>>> np.matmul(a,b)[0,1,1]
98
>>> sum(a[0,:,:] * b[:, :, 1])
98
```

Vector, vector returns the scalar inner product, but neither argument is [complex](#)-conjugated:

```
>>> np.matmul([2j, 3j], [2j, 3j])
(-13+0j)
```

Scalar multiplication raises an error.

```
>>> np.matmul([1,2], 3)
Traceback (most recent call last):
...
ValueError: Scalar operands are not allowed, use '*' instead
```

**maximum\_sctype(t)**  
Return the scalar type of highest precision of the same kind as the input.

```
Parameters

t : dtype or dtype specifier
 The input data type. This can be a `dtype` object or an object that
 is convertible to a `dtype``.

Returns

out : dtype
 The highest precision data type of the same kind \\(`dtype`.kind`\\\) as `t`.

See Also

obj2sctype, mintypecode, sctype2char
dtype

Examples

>>> np.maximum_sctype\\\(np.int\\\)
<type 'numpy.int64'maximum_sctype\\\(np.uint8\\\)
<type 'numpy.uint64'maximum_sctype\\\(np.complex\\\)
<type 'numpy.complex192'>

>>> np.maximum_sctype\\\(str\\\)
<type 'numpy.string_'maximum_sctype\\\('i2'\\\)
<type 'numpy.int64'maximum_sctype\\\('f4'\\\)
<type 'numpy.float96'>

may_share_memory\\\(...\\\)
may_share_memory\\\(a, b, max_work=None\\\)

Determine if two arrays might share memory

A return of True does not necessarily mean that the two arrays
share any element. It just means that they *might*.

Only the memory bounds of a and b are checked by default.

Parameters

a, b : ndarray
 Input arrays
max_work : int, optional
 Effort to spend on solving the overlap problem. See
 `shares_memory` for details. Default for ``may_share_memory``
 is to do a bounds check.

Returns

out : bool

See Also

shares_memory

Examples

>>> np.may_share_memory\\\(np.array\\\(\\\[\\\[1,2\\\]\\\]\\\), np.array\\\(\\\[\\\[5,8,9\\\]\\\]\\\)\\\)
False
>>> x = np.zeros\\\(\\\[3, 4\\\]\\\)
>>> np.may_share_memory\\\(x\\\[:,0\\\], x\\\[:,1\\\]\\\)
True

mean\\\(a, axis=None, dtype=None, out=None, keepdims=False\\\)
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over
the flattened array by default, otherwise over the specified axis.
`float64` intermediate and return values are used for integer inputs.

Parameters

a : array_like
 Array containing numbers whose mean is desired. If `a` is not an
 array, a conversion is attempted.
axis : None or int or tuple of ints, optional
 Axis or axes along which the means are computed. The default is to
 compute the mean of the flattened array.

.. versionadded: 1.7.0

If this is a tuple of ints, a mean is performed over multiple axes,
instead of a single axis or all the axes as before.
```

**dtype** : data-type, optional  
 Type to use in computing the mean. For `integer` inputs, the default is `'float64'`; for `floating` point inputs, it is the same as the input `dtype`.

**out** : `ndarray`, optional  
 Alternate output array in which to place the result. The default is `'None'`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary.  
 See `'doc.ufuncs'` for details.

**keepdims** : bool, optional  
 If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will `broadcast` correctly against the original `'arr'`.

**Returns**  
`m` : `ndarray`, see `dtype` parameter above  
 If `'out=None'`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See Also**  
`average` : Weighted average  
`std`, `var`, `nanmean`, `nanstd`, `nanvar`

**Notes**  
 The arithmetic mean is the sum of the elements along the axis divided by the `number` of elements.

Note that for `floating`-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `'float32'` (see example below). Specifying a higher-precision accumulator using the `'dtype'` keyword can alleviate this issue.

**Examples**

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])

In single precision, 'mean' can be inaccurate:
```

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.546875

Computing the mean in float64 is more accurate:
```

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

**median(a, axis=None, out=None, overwrite\_input=False, keepdims=False)**  
 Compute the median along the specified axis.

Returns the median of the array elements.

**Parameters**

**a** : `array_like`  
 Input array or `object` that can be converted to an array.

**axis** : {`int`, sequence of `int`, `None`}, optional  
 Axis or axes along which the medians are computed. The default is to compute the median along a flattened version of the array.  
 A sequence of axes is supported since version 1.9.0.

**out** : `ndarray`, optional  
 Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite\_input** : bool, optional  
 If True, then allow use of memory of input array `'a'` for calculations. The input array will be modified by the call to `'median'`. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. If `'overwrite_input'` is `'True'` and `'a'` is not already an `'ndarray'`, an error will be raised.

**keepdims** : bool, optional  
 If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will `broadcast` correctly against the original `'arr'`.

```
.. versionadded:: 1.9.0

>Returns

median : ndarray
 A new array holding the result. If the input contains integers
 or floats smaller than ``float64``, then the output data-type is
 ``np.float64``. Otherwise, the data-type of the output is the
 same as that of the input. If `out` is specified, that array is
 returned instead.

See Also

mean, percentile

Notes

Given a vector ``V`` of length ``N``, the median of ``V`` is the
middle value of a sorted copy of ``V``, ``V_sorted`` - i
e., ``V_sorted[(N-1)/2]`` when ``N`` is odd, and the average of the
two middle values of ``V_sorted`` when ``N`` is even.

Examples

>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10, 7, 4],
 [3, 2, 1]])
>>> np.median(a)
3.5
>>> np.median(a, axis=0)
array([6.5, 4.5, 2.5])
>>> np.median(a, axis=1)
array([7., 2.])
>>> m = np.median(a, axis=0)
>>> out = np.zeros_like(m)
>>> np.median(a, axis=0, out=m)
array([6.5, 4.5, 2.5])
>>> m
array([6.5, 4.5, 2.5])
>>> b = a.copy()
>>> np.median(b, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.median(b, axis=None, overwrite_input=True)
3.5
>>> assert not np.all(a==b)

meshgrid(*xi, **kwargs)
Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of
N-D scalar/vector fields over N-D grids, given
one-dimensional coordinate arrays x1, x2,..., xn.

.. versionchanged:: 1.9
 1-D and 0-D cases are allowed.

Parameters

x1, x2,..., xn : array_like
 1-D arrays representing the coordinates of a grid.
indexing : {'xy', 'ij'}, optional
 Cartesian ('xy', default) or matrix ('ij') indexing of output.
 See Notes for more details.

 .. versionadded:: 1.7.0
sparse : bool, optional
 If True a sparse grid is returned in order to conserve memory.
 Default is False.

 .. versionadded:: 1.7.0
copy : bool, optional
 If False, a view into the original arrays are returned in order to
 conserve memory. Default is True. Please note that
 ``sparse=False, copy=False`` will likely return non-contiguous
 arrays. Furthermore, more than one element of a broadcast array
 may refer to a single memory location. If you need to write to the
 arrays, make copies first.

 .. versionadded:: 1.7.0

>Returns

X1, X2,..., XN : ndarray
 For vectors `x1`, `x2`,..., `xn` with lengths ``Ni=len(xi)``,
 return ``{(N1, N2, N3,...Nn)}`` shaped arrays if indexing='ij'
 or ``{(N2, N1, N3,...Nn)}`` shaped arrays if indexing='xy'
```

with the elements of `xi` repeated to fill the [matrix](#) along the first dimension for `x1`, the second for `x2` and so on.

**Notes**

-----  
This function supports both indexing conventions through the `indexing` keyword argument. Giving the string 'ij' returns a meshgrid with [matrix](#) indexing, while 'xy' returns a meshgrid with Cartesian indexing. In the 2-D case with inputs of length M and N, the outputs are of shape (N, M) for 'xy' indexing and (M, N) for 'ij' indexing. In the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for 'xy' indexing and (M, N, P) for 'ij' indexing. The difference is illustrated by the following code snippet::

```
xv, yv = meshgrid(x, y, sparse=False, indexing='ij')
for i in range(nx):
 for j in range(ny):
 # treat xv[i,j], yv[i,j]

xv, yv = meshgrid(x, y, sparse=False, indexing='xy')
for i in range(nx):
 for j in range(ny):
 # treat xv[j,i], yv[j,i]
```

In the 1-D and 0-D case, the indexing and sparse keywords have no effect.

**See Also**

-----  
[index\\_tricks.mgrid](#) : Construct a multi-dimensional "meshgrid" using indexing notation.  
[index\\_tricks.ogrid](#) : Construct an open multi-dimensional "meshgrid" using indexing notation.

**Examples**

```

>>> nx, ny = (3, 2)
>>> x = np.linspace(0, 1, nx)
>>> y = np.linspace(0, 1, ny)
>>> xv, yv = meshgrid(x, y)
>>> xv
array([[0. , 0.5, 1.],
 [0. , 0.5, 1.]])
>>> yv
array([[0., 0., 0.],
 [1., 1., 1.]])
>>> xv, yv = meshgrid(x, y, sparse=True) # make sparse output arrays
>>> xv
array([[0. , 0.5, 1.]])
>>> yv
array([[0.],
 [1.]])
`meshgrid` is very useful to evaluate functions on a grid.
```

```
>>> x = np.arange(-5, 5, 0.1)
>>> y = np.arange(-5, 5, 0.1)
>>> xx, yy = meshgrid(x, y, sparse=True)
>>> z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)
>>> h = plt.contourf(x,y,z)
```

**[min\\_scalar\\_type](#)(...)**

[min\\_scalar\\_type](#)(a)  
For scalar ``a``, returns the data type with the smallest size and smallest scalar kind which can hold its value. For non-scalar array ``a``, returns the vector's [dtype](#) unmodified.

Floating point values are not demoted to integers, and [complex](#) values are not demoted to floats.

**Parameters**

-----  
a : scalar or array\_like  
The value whose minimal data type is to be found.

**Returns**

-----  
out : [dtype](#)  
The minimal data type.

**Notes**

-----  
.. versionadded:: 1.6.0

**See Also**

-----  
[result\\_type](#), [promote\\_types](#), [dtype](#), [can\\_cast](#)

**Examples**

```

>>> np.min_scalar_type(10)
dtype('uint8')

>>> np.min_scalar_type(-260)
dtype('int16')

>>> np.min_scalar_type(3.1)
dtype('float16')

>>> np.min_scalar_type(1e50)
dtype('float64')

>>> np.min_scalar_type(np.arange(4,dtype='f8'))
dtype('float64')

mintypecode(typechars, typeset='GDFgdf', default='d')
 Return the character for the minimum-size type to which given types can
 be safely cast.

 The returned type character must represent the smallest size dtype such
 that an array of the returned type can handle the data from an array of
 all types in `typechars` (or if `typechars` is an array, then its
 dtype.char).

 Parameters

 typechars : list of str or array_like
 If a list of strings, each string should represent a dtype.
 If array_like, the character representation of the array dtype is used.
 typeset : str, optional
 The set of characters that the returned character is chosen from.
 The default set is 'GDFgdf'.
 default : str, optional
 The default character, this is returned if none of the characters in
 `typechars` matches a character in `typeset`.

 Returns

 typechar : str
 The character representing the minimum-size type that was found.

 See Also

 dtype, sctype2char, maximum_sctype

 Examples

 >>> np.mintypecode(['d', 'f', 'S'])
'd'
 >>> x = np.array([1.1, 2-3.j])
 >>> np.mintypecode(x)
'D'

 >>> np.mintypecode('abceh', default='G')
'G'

mirr(values, finance_rate, reinvest_rate)
 Modified internal rate of return.

 Parameters

 values : array_like
 Cash flows (must contain at least one positive and one negative
 value) or nan is returned. The first value is considered a sunk
 cost at time zero.
 finance_rate : scalar
 Interest rate paid on the cash flows
 reinvest_rate : scalar
 Interest rate received on the cash flows upon reinvestment

 Returns

 out : float
 Modified internal rate of return

moveaxis(a, source, destination)
 Move axes of an array to new positions.

 Other axes remain in their original order.

 ... versionadded::1.11.0

 Parameters

 a : np.ndarray
 The array whose axes should be reordered.
 source : int or sequence of int

```

```
Original positions of the axes to move. These must be unique.
destination : int or sequence of int
 Destination positions for each of the original axes. These must also be
 unique.

Returns

result : np.ndarray
 Array with moved axes. This array is a view of the input array.

See Also

transpose: Permute the dimensions of an array.
swapaxes: Interchange two axes of an array.

Examples

>>> x = np.zeros((3, 4, 5))
>>> np.moveaxis(x, 0, -1).shape
(4, 5, 3)
>>> np.moveaxis(x, -1, 0).shape
(5, 3, 4)

These all achieve the same result:

>>> np.transpose(x).shape
(5, 4, 3)
>>> np.swapaxis(x, 0, -1).shape
(5, 4, 3)
>>> np.moveaxis(x, [0, 1], [-1, -2]).shape
(5, 4, 3)
>>> np.moveaxis(x, [0, 1, 2], [-1, -2, -3]).shape
(5, 4, 3)

msort\(a\)
Return a copy of an array sorted along the first axis.

Parameters

a : array_like
 Array to be sorted.

Returns

sorted_array : ndarray
 Array of the same type and shape as `a`.

See Also

sort

Notes

``np.msort(a)`` is equivalent to ``np.sort(a, axis=0)``.

nan_to_num\(x\)
Replace nan with zero and inf with finite numbers.

Returns an array or scalar replacing Not a Number (NaN) with zero,
(positive) infinity with a very large number and negative infinity
with a very small (or negative) number.

Parameters

x : array_like
 Input data.

Returns

out : ndarray
 New Array with the same shape as `x` and dtype of the element in
 `x` with the greatest precision. If `x` is inexact, then NaN is
 replaced by zero, and infinity (-infinity) is replaced by the
 largest (smallest or most negative) floating point value that fits
 in the output dtype. If `x` is not inexact, then a copy of `x` is
 returned.

See Also

isinf : Shows which elements are negative or positive infinity.
isnan : Shows which elements are negative infinity.
isneginf : Shows which elements are positive infinity.
isnan : Shows which elements are Not a Number (NaN).
isfinite : Shows which elements are finite (not NaN, not infinity)

Notes

```

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

#### Examples

```
>>> np.set_printoptions(precision=8)
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
>>> np.nan_to_num(x)
array([1.79769313e+308, -1.79769313e+308, 0.00000000e+000,
 -1.28000000e+002, 1.28000000e+002])
```

### **nanargmax(a, axis=None)**

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slices ``ValueError`` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and -Infs.

#### Parameters

```
a : array_like
 Input data.
axis : int, optional
 Axis along which to operate. By default flattened input is used.

Returns

index_array : ndarray
 An array of indices or a single index value.
```

#### See Also

```
argmax, nanargmin
```

#### Examples

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmax(a)
0
>>> np.nanargmax(a)
1
>>> np.nanargmax(a, axis=0)
array([1, 0])
>>> np.nanargmax(a, axis=1)
array([1, 1])
```

### **nanargmin(a, axis=None)**

Return the indices of the minimum values in the specified axis ignoring NaNs. For all-NaN slices ``ValueError`` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and Infs.

#### Parameters

```
a : array_like
 Input data.
axis : int, optional
 Axis along which to operate. By default flattened input is used.

Returns

index_array : ndarray
 An array of indices or a single index value.
```

#### See Also

```
argmin, nanargmax
```

#### Examples

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmin(a)
0
>>> np.nanargmin(a)
2
>>> np.nanargmin(a, axis=0)
array([1, 1])
>>> np.nanargmin(a, axis=1)
array([1, 0])
```

### **nanmax(a, axis=None, out=None, keepdims=False)**

Return the maximum of an array or maximum along an axis, ignoring any NaNs. When all-NaN slices are encountered a ``RuntimeWarning`` is raised and NaN is returned for that slice.

#### Parameters

```
a : array_like
 Array containing numbers whose maximum is desired. If `a` is not an
```

```

 array, a conversion is attempted.
axis : int, optional
 Axis along which the maximum is computed. The default is to compute
 the maximum of the flattened array.
out : ndarray, optional
 Alternate output array in which to place the result. The default
 is ``None``; if provided, it must have the same shape as the
 expected output, but the type will be cast if necessary. See
 `doc.ufuncs` for details.

.. versionadded:: 1.8.0
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in the
 result as dimensions with size one. With this option, the result
 will broadcast correctly against the original `a`.

.. versionadded:: 1.8.0

>Returns

nanmax : ndarray
 An array with the same shape as `a`, with the specified axis removed.
 If `a` is a 0-d array, or if axis is None, an ndarray scalar is
 returned. The same dtype as `a` is returned.

See Also

nanmin :
 The minimum value of an array along a given axis, ignoring any NaNs.
amax :
 The maximum value of an array along a given axis, propagating any NaNs.
fmax :
 Element-wise maximum of two arrays, ignoring any NaNs.
maximum :
 Element-wise maximum of two arrays, propagating any NaNs.
isnan :
 Shows which elements are Not a Number (NaN).
isfinite:
 Shows which elements are neither NaN nor infinity.

amin, fmin, minimum

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
(IEEE 754). This means that Not a Number is not equivalent to infinity.
Positive infinity is treated as a very large number and negative
infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.max.

Examples

>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmax(a)
3.0
>>> np.nanmax(a, axis=0)
array([3., 2.])
>>> np.nanmax(a, axis=1)
array([2., 3.])

When positive infinity and negative infinity are present:

>>> np.nanmax([1, 2, np.nan, np.NINF])
2.0
>>> np.nanmax([1, 2, np.nan, np.inf])
inf

nanmean(a, axis=None, dtype=None, out=None, keepdims=False)
Compute the arithmetic mean along the specified axis, ignoring NaNs.

Returns the average of the array elements. The average is taken over
the flattened array by default, otherwise over the specified axis.
`float64` intermediate and return values are used for integer inputs.

For all-NaN slices, NaN is returned and a `RuntimeWarning` is raised.

.. versionadded:: 1.8.0

Parameters

a : array_like
 Array containing numbers whose mean is desired. If `a` is not an
 array, a conversion is attempted.
axis : int, optional
 Axis along which the means are computed. The default is to compute
 the mean of the flattened array.
dtype : data-type, optional
 Type to use in computing the mean. For integer inputs, the default

```

```

 is `float64`; for inexact inputs, it is the same as the input
dtype.
out : ndarray, optional
 Alternative output array in which to place the result. The default
 is ``None``; if provided, it must have the same shape as the
 expected output, but the type will be cast if necessary. See
 'doc.ufuncs' for details.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in the
 result as dimensions with size one. With this option, the result
 will broadcast correctly against the original 'arr'.

```

Returns

```

m : ndarray, see dtype parameter above
 If 'out=None', returns a new array containing the mean values,
 otherwise a reference to the output array is returned. Nan is
 returned for slices that contain only NaNs.

```

See Also

```

average : Weighted average
mean : Arithmetic mean taken while not ignoring NaNs
var, nanvar

```

Notes

```

The arithmetic mean is the sum of the non-NaN elements along the axis
divided by the number of non-NaN elements.

Note that for floating-point input, the mean is computed using the same
precision the input has. Depending on the input data, this can cause
the results to be inaccurate, especially for float32. Specifying a
higher-precision accumulator using the 'dtype' keyword can alleviate
this issue.

```

Examples

```

>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanmean(a)
2.6666666666666665
>>> np.nanmean(a, axis=0)
array([2., 4.])
>>> np.nanmean(a, axis=1)
array([1., 3.5])

```

**nanmedian**(a, axis=None, out=None, overwrite\_input=False, keepdims=False)

Compute the median along the specified axis, while ignoring NaNs.

Returns the median of the array elements.

.. versionadded:: 1.9.0

Parameters

```

a : array_like
 Input array or object that can be converted to an array.
axis : {int, sequence of int, None}, optional
 Axis or axes along which the medians are computed. The default
 is to compute the median along a flattened version of the array.
 A sequence of axes is supported since version 1.9.0.
out : ndarray, optional
 Alternative output array in which to place the result. It must
 have the same shape and buffer length as the expected output,
 but the type (of the output) will be cast if necessary.
overwrite_input : bool, optional
 If True, then allow use of memory of input array 'a' for
 calculations. The input array will be modified by the call to
 'median'. This will save memory when you do not need to preserve
 the contents of the input array. Treat the input as undefined,
 but it will probably be fully or partially sorted. Default is
 False. If 'overwrite_input' is ``True`` and 'a' is not already an
 ndarray, an error will be raised.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in
 the result as dimensions with size one. With this option, the
 result will broadcast correctly against the original 'arr'.

```

Returns

```

median : ndarray
 A new array holding the result. If the input contains integers
 or floats smaller than ``float64``, then the output data-type is
 ``np.float64``. Otherwise, the data-type of the output is the
 same as that of the input. If 'out' is specified, that array is
 returned instead.

```

See Also

```

mean, median, percentile

Notes

Given a vector ``V`` of length ``N``, the median of ``V`` is the
middle value of a sorted copy of ``V``, ``V_sorted`` - i.e.,
``V_sorted[(N-1)/2]``, when ``N`` is odd and the average of the two
middle values of ``V_sorted`` when ``N`` is even.

Examples

>>> a = np.array([[10.0, 7, 4], [3, 2, 1]])
>>> a[0, 1] = np.nan
>>> a
array([[10., nan, 4.],
 [3., 2., 1.]])
>>> np.median(a)
nan
>>> np.nanmedian(a)
3.0
>>> np.nanmedian(a, axis=0)
array([6.5, 2., 2.5])
>>> np.median(a, axis=1)
array([7., 2.])
>>> b = a.copy()
>>> np.nanmedian(b, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.nanmedian(b, axis=None, overwrite_input=True)
3.0
>>> assert not np.all(a==b)

nanmin(a, axis=None, out=None, keepdims=False)
Return minimum of an array or minimum along an axis, ignoring any NaNs.
When all-NaN slices are encountered a ``RuntimeWarning`` is raised and
NaN is returned for that slice.

Parameters

a : array_like
 Array containing numbers whose minimum is desired. If `a` is not an
 array, a conversion is attempted.
axis : int, optional
 Axis along which the minimum is computed. The default is to compute
 the minimum of the flattened array.
out : ndarray, optional
 Alternate output array in which to place the result. The default
 is `None`; if provided, it must have the same shape as the
 expected output, but the type will be cast if necessary. See
 `doc.ufuncs` for details.

.. versionadded:: 1.8.0
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in the
 result as dimensions with size one. With this option, the result
 will broadcast correctly against the original `a`.

.. versionadded:: 1.8.0

Returns

nanmin : ndarray
 An array with the same shape as `a`, with the specified axis
 removed. If `a` is a 0-d array, or if axis is None, an ndarray
 scalar is returned. The same dtype as `a` is returned.

See Also

nanmax :
 The maximum value of an array along a given axis, ignoring any NaNs.
amin :
 The minimum value of an array along a given axis, propagating any NaNs.
fmin :
 Element-wise minimum of two arrays, ignoring any NaNs.
minimum :
 Element-wise minimum of two arrays, propagating any NaNs.
isnan :
 Shows which elements are Not a Number (NaN).
isfinite:
 Shows which elements are neither NaN nor infinity.

amax, fmax, maximum

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic
(IEEE 754). This means that Not a Number is not equivalent to infinity.
Positive infinity is treated as a very large number and negative

```

```

infinity is treated as a very small (i.e. negative) number.
If the input has a integer type the function is equivalent to np.min.

Examples

>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmin(a)
1.0
>>> np.nanmin(a, axis=0)
array([1., 2.])
>>> np.nanmin(a, axis=1)
array([1., 3.])

When positive infinity and negative infinity are present:

>>> np.nanmin([1, 2, np.nan, np.inf])
1.0
>>> np.nanmin([1, 2, np.nan, np.NINF])
-inf

nanpercentile(a, q, axis=None, out=None, overwrite_input=False, interpolation='linear', keepdims=False)
Compute the qth percentile of the data along the specified axis,
while ignoring nan values.

Returns the qth percentile(s) of the array elements.

.. versionadded:: 1.9.0

Parameters

a : array_like
 Input array or object that can be converted to an array.
q : float in range of [0,100] (or sequence of floats)
 Percentile to compute, which must be between 0 and 100
 inclusive.
axis : {int, sequence of int, None}, optional
 Axis or axes along which the percentiles are computed. The
 default is to compute the percentile(s) along a flattened
 version of the array. A sequence of axes is supported since
 version 1.9.0.
out : ndarray, optional
 Alternative output array in which to place the result. It must
 have the same shape and buffer length as the expected output,
 but the type (of the output) will be cast if necessary.
overwrite_input : bool, optional
 If True, then allow use of memory of input array `a` for
 calculations. The input array will be modified by the call to
 'percentile'. This will save memory when you do not need to
 preserve the contents of the input array. In this case you
 should not make any assumptions about the contents of the input
 `a` after this function completes -- treat it as undefined.
 Default is False. If `a` is not already an array, this parameter
 will have no effect as `a` will be converted to an array
 internally regardless of the value of this parameter.
interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}
 This optional parameter specifies the interpolation method to
 use when the desired quantile lies between two data points
 ``i < j``:
 * linear: ``i + (j - i) * fraction``, where ``fraction`` is
 the fractional part of the index surrounded by ``i`` and
 ``j``.
 * lower: ``i``.
 * higher: ``j``.
 * nearest: ``i`` or ``j``, whichever is nearest.
 * midpoint: ``(i + j) / 2``.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in
 the result as dimensions with size one. With this option, the
 result will broadcast correctly against the original array `a`.

Returns

percentile : scalar or ndarray
 If `q` is a single percentile and `axis=None`, then the result
 is a scalar. If multiple percentiles are given, first axis of
 the result corresponds to the percentiles. The other axes are
 the axes that remain after the reduction of `a`. If the input
 contains integers or floats smaller than ``float64````, the output
 data-type is ``float64``. Otherwise, the output data-type is the
 same as that of the input. If `out` is specified, that array is
 returned instead.

See Also

nanmean, nanmedian, percentile, median, mean

Notes

```

Given a vector ``V`` of length ``N``, the ``q``-th percentile of ``V`` is the value ``q/100`` of the way from the minimum to the maximum in a sorted copy of ``V``. The values and distances of the two nearest neighbors as well as the `interpolation` parameter will determine the percentile if the normalized ranking does not match the location of ``q`` exactly. This function is the same as the median if ``q=50``, the same as the minimum if ``q=0`` and the same as the maximum if ``q=100``.

#### Examples

```

>>> a = np.array([[10., 7., 4.], [3., 2., 1.]])
>>> a[0][1] = np.nan
>>> a
array([[10., nan, 4.],
 [3., 2., 1.]])
>>> np.percentile(a, 50)
nan
>>> np.nanpercentile(a, 50)
3.5
>>> np.nanpercentile(a, 50, axis=0)
array([6.5, 2., 2.5])
>>> np.nanpercentile(a, 50, axis=1, keepdims=True)
array([[7.],
 [2.]])
>>> m = np.nanpercentile(a, 50, axis=0)
>>> out = np.zeros_like(m)
>>> np.nanpercentile(a, 50, axis=0, out=out)
array([6.5, 2., 2.5])
>>> m
array([6.5, 2., 2.5])

>>> b = a.copy()
>>> np.nanpercentile(b, 50, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a==b)
```

#### nanprod(a, axis=None, dtype=None, out=None, keepdims=0)

Return the product of array elements over a given axis treating Not a Numbers (NaNs) as zero.

One is returned for slices that are all-NaN or empty.

.. versionadded:: 1.10.0

#### Parameters

```

a : array_like
 Array containing numbers whose sum is desired. If `a` is not an
 array, a conversion is attempted.
axis : int, optional
 Axis along which the product is computed. The default is to compute
 the product of the flattened array.
dtype : data-type, optional
 The type of the returned array and of the accumulator in which the
 elements are summed. By default, the `dtype` of `a` is used. An
 exception is when `a` has an integer type with less precision than
 the platform (intp). In that case, the default will be either
 (int32) or (int64) depending on whether the platform is 32 or 64
 bits. For inexact inputs, `dtype` must be inexact.
out : ndarray, optional
 Alternate output array in which to place the result. The default
 is ``None``. If provided, it must have the same shape as the
 expected output, but the type will be cast if necessary. See
 'doc.ufuncs' for details. The casting of NaN to integer can yield
 unexpected results.
keepdims : bool, optional
 If True, the axes which are reduced are left in the result as
 dimensions with size one. With this option, the result will
 broadcast correctly against the original 'arr'.
```

#### Returns

```

y : ndarray or numpy scalar
```

#### See Also

```

numpy.prod : Product across array propagating NaNs.
isnan : Show which elements are NaN.
```

#### Notes

```

NumPy integer arithmetic is modular. If the size of a product exceeds
the size of an integer accumulator, its value will wrap around and the
result will be incorrect. Specifying ``dtype=double`` can alleviate
that problem.
```

#### Examples

```
>>> np.nanprod(1)
1
>>> np.nanprod([1])
1
>>> np.nanprod([1, np.nan])
1.0
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanprod(a)
6.0
>>> np.nanprod(a, axis=0)
array([3., 2.])

nanstd(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
 Compute the standard deviation along the specified axis, while
 ignoring NaNs.

 Returns the standard deviation, a measure of the spread of a
 distribution, of the non-NaN array elements. The standard deviation is
 computed for the flattened array by default, otherwise over the
 specified axis.

 For all-NaN slices or slices with zero degrees of freedom, NaN is
 returned and a 'RuntimeWarning' is raised.

.. versionadded:: 1.8.0

Parameters

a : array_like
 Calculate the standard deviation of the non-NaN values.
axis : int, optional
 Axis along which the standard deviation is computed. The default is
 to compute the standard deviation of the flattened array.
dtype : dtype, optional
 Type to use in computing the standard deviation. For arrays of
 integer type the default is float64, for arrays of float types it
 is the same as the array type.
out : ndarray, optional
 Alternative output array in which to place the result. It must have
 the same shape as the expected output but the type (of the
 calculated values) will be cast if necessary.
ddof : int, optional
 Means Delta Degrees of Freedom. The divisor used in calculations
 is ``N - ddof``, where ``N`` represents the number of non-NaN
 elements. By default `ddof` is zero.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left
 in the result as dimensions with size one. With this option,
 the result will broadcast correctly against the original `arr`.

Returns

standard_deviation : ndarray, see dtype parameter above.
 If 'out' is None, return a new array containing the standard
 deviation, otherwise return a reference to the output array. If
 ddof is >= the number of non-NaN elements in a slice or the slice
 contains only NaNs, then the result for that slice is NaN.

See Also

var, mean, std
nanvar, nanmean
numpy.doc.ufuncs : Section "Output arguments"

Notes

The standard deviation is the square root of the average of the squared
deviations from the mean: ``std = sqrt(abs\(x - x.mean\(\)\)**2\)\)``.

The average squared deviation is normally calculated as
``x.sum\(\) / N``, where ``N = len\(x\)``. If, however, `ddof` is
specified, the divisor ``N - ddof`` is used instead. In standard
statistical practice, ``ddof=1`` provides an unbiased estimator of the
variance of the infinite population. ``ddof=0`` provides a maximum
likelihood estimate of the variance for normally distributed variables.
The standard deviation computed in this function is the square root of
the estimated variance, so even with ``ddof=1``, it will not be an
unbiased estimate of the standard deviation per se.

Note that, for complex numbers, `std` takes the absolute value before
squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same
precision the input has. Depending on the input data, this can cause
the results to be inaccurate, especially for float32 \(see example
below\). Specifying a higher-accuracy accumulator using the `dtype`
keyword can alleviate this issue.

Examples
```

```

>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanstd(a)
1.247219128924647
>>> np.nanstd(a, axis=0)
array([1., 0.])
>>> np.nanstd(a, axis=1)
array([0., 0.5])

nansum(a, axis=None, dtype=None, out=None, keepdims=0)
Return the sum of array elements over a given axis treating Not a
Numbers (NaNs) as zero.

In Numpy versions <= 1.8 Nan is returned for slices that are all-NaN or
empty. In later versions zero is returned.

Parameters

a : array_like
 Array containing numbers whose sum is desired. If `a` is not an
 array, a conversion is attempted.
axis : int, optional
 Axis along which the sum is computed. The default is to compute the
 sum of the flattened array.
dtype : data-type, optional
 The type of the returned array and of the accumulator in which the
 elements are summed. By default, the dtype of `a` is used. An
 exception is when `a` has an integer type with less precision than
 the platform (u)intp. In that case, the default will be either
 (u)int32 or (u)int64 depending on whether the platform is 32 or 64
 bits. For inexact inputs, dtype must be inexact.
 .. versionadded:: 1.8.0
out : ndarray, optional
 Alternate output array in which to place the result. The default
 is ``None``. If provided, it must have the same shape as the
 expected output, but the type will be cast if necessary. See
 `doc.ufuncs` for details. The casting of NaN to integer can yield
 unexpected results.
 .. versionadded:: 1.8.0
keepdims : bool, optional
 If True, the axes which are reduced are left in the result as
 dimensions with size one. With this option, the result will
 broadcast correctly against the original `arr`.
 .. versionadded:: 1.8.0

Returns

y : ndarray or numpy scalar

See Also

numpy.sum : Sum across array propagating NaNs.
isnan : Show which elements are NaN.
isfinite : Show which elements are not NaN or +/-inf.

Notes

If both positive and negative infinity are present, the sum will be Not
A Number (NaN).

Numpy integer arithmetic is modular. If the size of a sum exceeds the
size of an integer accumulator, its value will wrap around and the
result will be incorrect. Specifying ``dtype=double`` can alleviate
that problem.

Examples

>>> np.nansum(1)
1
>>> np.nansum([1])
1
>>> np.nansum([1, np.nan])
1.0
>>> a = np.array([[1, 1], [1, np.nan]])
>>> np.nansum(a)
3.0
>>> np.nansum(a, axis=0)
array([2., 1.])
>>> np.nansum([1, np.nan, np.inf])
inf
>>> np.nansum([1, np.nan, np.NINF])
-nan
>>> np.nansum([1, np.nan, np.inf, -np.inf]) # both +/- infinity present
nan
```

```

nanvar(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
 Compute the variance along the specified axis, while ignoring NaNs.

 Returns the variance of the array elements, a measure of the spread of
 a distribution. The variance is computed for the flattened array by
 default, otherwise over the specified axis.

 For all-NaN slices or slices with zero degrees of freedom, NaN is
 returned and a `RuntimeWarning` is raised.

.. versionadded:: 1.8.0

Parameters

a : array_like
 Array containing numbers whose variance is desired. If `a` is not an
 array, a conversion is attempted.
axis : int, optional
 Axis along which the variance is computed. The default is to compute
 the variance of the flattened array.
dtype : data-type, optional
 Type to use in computing the variance. For arrays of integer type
 the default is float32; for arrays of float types it is the same as
 the array type.
out : ndarray, optional
 Alternate output array in which to place the result. It must have
 the same shape as the expected output, but the type is cast if
 necessary.
ddof : int, optional
 "Delta Degrees of Freedom": the divisor used in the calculation is
 ``N - ddof``, where ``N`` represents the number of non-NaN
 elements. By default `ddof` is zero.
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left
 in the result as dimensions with size one. With this option,
 the result will broadcast correctly against the original `arr`.

Returns

variance : ndarray, see dtype parameter above
 If `out` is None, return a new array containing the variance,
 otherwise return a reference to the output array. If ddof is >= the
 number of non-NaN elements in a slice or the slice contains only
 NaNs, then the result for that slice is NaN.

See Also

std : Standard deviation
mean : Average
var : Variance while not ignoring NaNs
nanstd, nanmean
numpy.doc.ufuncs : Section "Output arguments"

Notes

The variance is the average of the squared deviations from the mean,
i.e., ``var = mean(abs(x - x.mean())**2)``.

The mean is normally calculated as ``x.sum() / N``, where ``N = len(x)``.
If, however, `ddof` is specified, the divisor ``N - ddof`` is used
instead. In standard statistical practice, ``ddof=1`` provides an
unbiased estimator of the variance of a hypothetical infinite
population. ``ddof=0`` provides a maximum likelihood estimate of the
variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before
squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same
precision the input has. Depending on the input data, this can cause
the results to be inaccurate, especially for float32 (see example
below). Specifying a higher-accuracy accumulator using the ```dtype``
keyword can alleviate this issue.

Examples

>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.var(a)
1.555555555555554
>>> np.nanvar(a, axis=0)
array([1., 0.])
>>> np.nanvar(a, axis=1)
array([0., 0.25])

ndfromtxt(fname, **kwargs)
 Load ASCII data stored in a file and return it as a single array.

Parameters

```

```
fname, kwargs : For a description of input parameters, see `genfromtxt`.
See Also

numpy.genfromtxt : generic function.
ndim(a)
 Return the number of dimensions of an array.

Parameters

a : array_like
 Input array. If it is not already an ndarray, a conversion is attempted.

Returns

number_of_dimensions : int
 The number of dimensions in `a`. Scalars are zero-dimensional.

See Also

ndarray.ndim : equivalent method
shape : dimensions of array
ndarray.shape : dimensions of array

Examples

>>> np.ndim([[1,2,3],[4,5,6]])
2
>>> np.ndim(np.array([[1,2,3],[4,5,6]]))
2
>>> np.ndim(1)
0
nested_iters(...)
newbuffer(...)
 newbuffer(size)

 Return a new uninitialized buffer object.

Parameters

size : int
 Size in bytes of returned buffer object.

Returns

newbuffer : buffer object
 Returned, uninitialized buffer object of `size` bytes.

nonzero(a)
 Return the indices of the elements that are non-zero.

 Returns a tuple of arrays, one for each dimension of `a`, containing the indices of the non-zero elements in that dimension. The values in `a` are always tested and returned in row-major, C-style order. The corresponding non-zero values can be obtained with::

 a[nonzero\(a\)\]

 To group the indices by element, rather than dimension, use::

 transpose\\(nonzero\\(a\\)\\)

 The result of this is always a 2-D array, with a row for each non-zero element.

Parameters

a : array_like
 Input array.

Returns

tuple_of_arrays : tuple
 Indices of elements that are non-zero.

See Also

flatnonzero :
 Return indices that are non-zero in the flattened version of the input array.
ndarray.nonzero :
 Equivalent ndarray method.
```

```

count_nonzero :
 Counts the number of non-zero elements in the input array.

Examples

>>> x = np.eye(3)
>>> x
array([[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]])
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))
>>> x[np.nonzero(x)]
array([1., 1., 1.])
>>> np.transpose(np.nonzero(x))
array([[0, 0],
 [1, 1],
 [2, 2]])

```

A common use for ``nonzero`` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, np.nonzero(a > 3) yields the indices of the `a` where the condition is true.

```

>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
array([[False, False, False],
 [True, True, True],
 [True, True, True]], dtype=bool)
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

```

The ``nonzero`` method of the boolean array can also be called.

```

>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

```

### **nper(rate, pmt, pv, fv=0, when='end')**

Compute the [number](#) of periodic payments.

```

Parameters

rate : array_like
 Rate of interest (per period)
pmt : array_like
 Payment
pv : array_like
 Present value
fv : array_like, optional
 Future value
when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
 When payments are due ('begin' (1) or 'end' (0))


```

#### **Notes**

The [number](#) of periods ``nper`` is computed by solving the equation::

$$fv + pv*(1+rate)^{**nper} + pmt*(1+rate*when)/rate*((1+rate)^{**nper}-1) = 0$$

but if ``rate = 0`` then::

$$fv + pv + pmt*nper = 0$$

#### **Examples**

If you only had \$150/month to pay towards the loan, how long would it take to pay-off a loan of \$8,000 at 7% annual interest?

```

>>> print(round(np.nper(0.07/12, -150, 8000), 5))
64.07335

```

So, over 64 months would be required to pay off the loan.

The same analysis could be done with several different interest rates and/or payments and/or total amounts to produce an entire table.

```

>>> np.nper(*(np.ogrid[0.07/12: 0.08/12: 0.01/12,
... -150 : 99 : 50 ,
... 8000 : 9001 : 1000]))
array([[[64.07334877, 74.06368256],
 [108.07548412, 127.99022654]],
 [[66.12443902, 76.87897353],
 [114.70165583, 137.90124779]])

```

### **npv(rate, values)**

Returns the NPV (Net Present Value) of a cash flow series.

**Parameters**  
**rate** : scalar  
     The discount rate.  
**values** : array\_like, [shape](#)(M, )  
     The values of the time series of cash flows. The (fixed) time interval between cash flow "events" must be the same as that for which 'rate' is given (i.e., if 'rate' is per year, then precisely a year is understood to elapse between each cash flow event). By convention, investments or "deposits" are negative, income or "withdrawals" are positive; 'values' must begin with the initial investment, thus 'values[0]' will typically be negative.

**Returns**  
**out** : [float](#)  
     The NPV of the input cash flow series 'values' at the discount 'rate'.

**Notes**  
**Returns** the result of: [G]  

$$\dots \text{math} :: \sum_{t=0}^{M-1} \frac{\text{values}_t}{(1+\text{rate})^t}$$

**References**  
... [G] L. J. Gitman, "Principles of Managerial Finance, Brief," 3rd ed., Addison-Wesley, 2003, pg. 346.

**Examples**  

$$>>> \text{np.npv}(0.281, [-100, 39, 59, 55, 20])$$
  
-0.0084785916384548798

(Compare with the Example given for `numpy.lib.financial.irr`)

**obj2sctype**(rep, default=None)  
Return the scalar [dtype](#) or NumPy equivalent of Python type of an [object](#).

**Parameters**  
**rep** : any  
     The [object](#) of which the type is returned.  
**default** : any, optional  
     If given, this is returned for objects whose types can not be determined. If not given, None is returned for those objects.

**Returns**  
**dtype** : [dtype](#) or Python type  
     The data type of 'rep'.

**See Also**  
**sctype2char**, **issctype**, **issubscotype**, **issubdtype**, **maximum\_sctype**

**Examples**  

$$>>> \text{np.obj2sctype}(\text{np.int32})$$
  
<type 'numpy.int32'>  

$$>>> \text{np.obj2sctype}(\text{np.array}([1., 2.]))$$
  
<type 'numpy.float64'>  

$$>>> \text{np.obj2sctype}(\text{np.array}([1.j]))$$
  
<type 'numpy.complex128'>

$$>>> \text{np.obj2sctype}(\text{dict})$$
  
<type 'numpy.object'\_>  

$$>>> \text{np.obj2sctype}(\text{'string'})$$
  
<type 'numpy.string'\_>
$$>>> \text{np.obj2sctype}(1, \text{default}=\text{list})$$
  
<type 'list'\_>

**ones**(shape, dtype=None, order='C')  
Return a new array of given shape and type, filled with ones.

**Parameters**  
**shape** : [int](#) or sequence of ints  
     Shape of the new array, e.g., `(2, 3)` or `2`.  
**dtype** : data-type, optional  
     The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.  
**order** : {'C', 'F'}, optional  
     Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns  
-----  
**out : ndarray**  
    Array of ones with the given shape, [dtype](#), and order.

See Also  
-----  
[zeros](#), [ones\\_like](#)

Examples  
-----  
**>>> np.ones(5)**  
**array([ 1., 1., 1., 1., 1.])**  
  
**>>> np.ones((5,), dtype=np.int)**  
**array([1, 1, 1, 1, 1])**  
  
**>>> np.ones((2, 1))**  
**array([[ 1.],**  
**[ 1.]])**  
  
**>>> s = (2,2)**  
**>>> np.ones(s)**  
**array([[ 1., 1.],**  
**[ 1., 1.]])**

**ones\_like(a, dtype=None, order='K', subok=True)**  
Return an array of ones with the same shape and type as a given array.

Parameters  
-----  
**a : array\_like**  
    The shape and data-type of `a` define these same attributes of  
    the returned array.  
**dtype : data-type, optional**  
    Overrides the data type of the result.  
  
 .. versionadded:: 1.6.0  
**order : {'C', 'F', 'A', or 'K'}, optional**  
    Overrides the memory layout of the result. 'C' means C-order,  
    'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,  
    'C' otherwise. 'K' means match the layout of `a` as closely  
    as possible.  
  
 .. versionadded:: 1.6.0  
**subok : bool, optional**  
    If True, then the newly created array will use the sub-class  
    type of 'a', otherwise it will be a base-class array. Defaults  
    to True.

Returns  
-----  
**out : ndarray**  
    Array of ones with the same shape and type as 'a'.

See Also  
-----  
[zeros\\_like](#) : Return an array of zeros with shape and type of input.  
[empty\\_like](#) : Return an empty array with shape and type of input.  
[zeros](#) : Return a new array setting values to zero.  
[ones](#) : Return a new array setting values to one.  
[empty](#) : Return a new uninitialized array.

Examples  
-----  
**>>> x = np.arange(6)**  
**>>> x = x.reshape((2, 3))**  
**>>> x**  
**array([[0, 1, 2],**  
**[3, 4, 5]])**  
**>>> np.ones\_like(x)**  
**array([[1, 1, 1],**  
**[1, 1, 1]])**  
  
**>>> y = np.arange(3, dtype=np.float)**  
**>>> y**  
**array([ 0., 1., 2.])**  
**>>> np.ones\_like(y)**  
**array([ 1., 1., 1.])**

**outer(a, b, out=None)**  
Compute the outer product of two vectors.

Given two vectors, ``a = [a<sub>0</sub>, a<sub>1</sub>, ..., a<sub>M</sub>]`` and  
``b = [b<sub>0</sub>, b<sub>1</sub>, ..., b<sub>N</sub>]``,  
the outer product [1]\_ is::  

$$[[a_0*b_0 \ a_0*b_1 \ ... \ a_0*b_N \ ]]$$

```
[a1*b0 .
[...
[aM*b0 aM*bN]]

Parameters

a : (M,) array_like
 First input vector. Input is flattened if
 not already 1-dimensional.
b : (N,) array_like
 Second input vector. Input is flattened if
 not already 1-dimensional.
out : (M, N) ndarray, optional
 A location where the result is stored

.. versionadded:: 1.9.0

Returns

out : (M, N) ndarray
 ``out[i, j] = a[i] * b[j]``

See also

inner, einsum

References

.. [1] : G. H. Golub and C. F. van Loan, *Matrix Computations*, 3rd
 ed., Baltimore, MD, Johns Hopkins University Press, 1996,
 pg. 8.

Examples

Make a (*very* coarse) grid for computing a Mandelbrot set:

>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[-2., -1., 0., 1., 2.],
 [-2., -1., 0., 1., 2.],
 [-2., -1., 0., 1., 2.],
 [-2., -1., 0., 1., 2.],
 [-2., -1., 0., 1., 2.]])
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
 [0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
 [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
 [0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
 [0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
>>> grid = rl + im
>>> grid
array([[-2.+2.j, -1.+2.j, 0.+2.j, 1.+2.j, 2.+2.j],
 [-2.+1.j, -1.+1.j, 0.+1.j, 1.+1.j, 2.+1.j],
 [-2.+0.j, -1.+0.j, 0.+0.j, 1.+0.j, 2.+0.j],
 [-2.-1.j, -1.-1.j, 0.-1.j, 1.-1.j, 2.-1.j],
 [-2.-2.j, -1.-2.j, 0.-2.j, 1.-2.j, 2.-2.j]])

An example using a "vector" of letters:

>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
 [b, bb, bbb],
 [c, cc, ccc]], dtype=object)

packbits(...)
packbits(myarray, axis=None)

Packs the elements of a binary-valued array into bits in a uint8 array.

The result is padded to full bytes by inserting zero bits at the end.

Parameters

myarray : array_like
 An integer type array whose elements should be packed to bits.
axis : int, optional
 The dimension over which bit-packing is done.
 ``None`` implies packing the flattened array.

Returns

packed : ndarray
 Array of type uint8 whose elements represent bits corresponding to the
 logical (0 or nonzero) value of the input elements. The shape of
 `packed` has the same number of dimensions as the input (unless `axis`
 is None, in which case the output is 1-D).
```

See Also

-----

unpackbits: Unpacks elements of a `uint8` array into a binary-valued output array.

Examples

-----

```
>>> a = np.array([[[1,0,1],
... [0,1,0]],
... [[1,1,0],
... [0,0,1]]])
>>> b = np.packbits(a, axis=-1)
>>> b
array([[160],[64],[192],[32]], dtype=uint8)
```

Note that in binary  $160 = 1010\ 0000$ ,  $64 = 0100\ 0000$ ,  $192 = 1100\ 0000$ , and  $32 = 0010\ 0000$ .

**pad**(array, pad\_width, mode, \*\*kwargs)

Pads an array.

Parameters

-----

array : array\_like of rank N  
Input array

pad\_width : {sequence, array\_like, int}  
Number of values padded to the edges of each axis.  
 $((before\_1, after\_1), \dots (before\_N, after\_N))$  unique pad widths for each axis.  
 $((before, after),)$  yields same before and after pad for each axis.  
(pad,) or int is a shortcut for before = after = pad width for all axes.

mode : str or function  
One of the following string values or a user supplied function.

- 'constant'  
Pads with a constant value.
- 'edge'  
Pads with the edge values of array.
- 'linear\_ramp'  
Pads with the linear ramp between end\_value and the array edge value.
- 'maximum'  
Pads with the maximum value of all or part of the vector along each axis.
- 'mean'  
Pads with the mean value of all or part of the vector along each axis.
- 'median'  
Pads with the median value of all or part of the vector along each axis.
- 'minimum'  
Pads with the minimum value of all or part of the vector along each axis.
- 'reflect'  
Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
- 'symmetric'  
Pads with the reflection of the vector mirrored along the edge of the array.
- 'wrap'  
Pads with the wrap of the vector along the axis.  
The first values are used to pad the end and the end values are used to pad the beginning.

<function>  
Padding function, see Notes.

stat\_length : sequence or int, optional  
Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value.

$((before\_1, after\_1), \dots (before\_N, after\_N))$  unique statistic lengths for each axis.

$((before, after),)$  yields same before and after statistic lengths for each axis.

(stat\_length,) or int is a shortcut for before = after = statistic length for all axes.

Default is ``None``, to use the entire axis.

constant\_values : sequence or int, optional  
Used in 'constant'. The values to set the padded values for each axis.

$((before\_1, after\_1), \dots (before\_N, after\_N))$  unique pad constants for each axis.

$((before, after),)$  yields same before and after constants for each

```

axis.

(constant,) or int is a shortcut for before = after = constant for
all axes.

Default is 0.

end_values : sequence or int, optional
 Used in 'linear_ramp'. The values used for the ending value of the
 linear_ramp and that will form the edge of the padded array.

((before_1, after_1), ... (before_N, after_N)) unique end values
for each axis.

((before, after),) yields same before and after end values for each
axis.

(constant,) or int is a shortcut for before = after = end value for
all axes.

Default is 0.

reflect_type : {'even', 'odd'}, optional
 Used in 'reflect', and 'symmetric'. The 'even' style is the
 default with an unaltered reflection around the edge value. For
 the 'odd' style, the extended part of the array is created by
 subtracting the reflected values from two times the edge value.

Returns

pad : ndarray
 Padded array of rank equal to `array` with shape increased
 according to `pad_width`.

Notes

.. versionadded:: 1.7.0

For an array with rank greater than 1, some of the padding of later
axes is calculated from padding of previous axes. This is easiest to
think about with a rank 2 array where the corners of the padded array
are calculated by using padded values from the first axis.

The padding function, if used, should return a rank 1 array equal in
length to the vector argument with padded values replaced. It has the
following signature::

 padding_func(vector, iaxis_pad_width, iaxis, **kwargs)

where

vector : ndarray
 A rank 1 array already padded with zeros. Padded values are
 vector[:pad_tuple[0]] and vector[-pad_tuple[1]:].
iaxis_pad_width : tuple
 A 2-tuple of ints, iaxis_pad_width[0] represents the number of
 values padded at the beginning of vector where
 iaxis_pad_width[1] represents the number of values padded at
 the end of vector.
iaxis : int
 The axis currently being calculated.
kwargs : misc
 Any keyword arguments the function requires.

Examples

>>> a = [1, 2, 3, 4, 5]
>>> np.lib.pad(a, (2,3), 'constant', constant_values=(4, 6))
array([4, 4, 1, 2, 3, 4, 5, 6, 6, 6])

>>> np.lib.pad(a, (2, 3), 'edge')
array([1, 1, 1, 2, 3, 4, 5, 5, 5])

>>> np.lib.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([5, 3, 1, 2, 3, 4, 5, 2, -1, -4])

>>> np.lib.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])

>>> np.lib.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])

>>> np.lib.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])

>>> a = [[1, 2], [3, 4]]
>>> np.lib.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
 [1, 1, 1, 2, 1, 1, 1],
 [1, 1, 1, 2, 1, 1, 1],
 [1, 1, 1, 2, 1, 1, 1],
 [3, 3, 3, 4, 3, 3, 3]])

```

```

[1, 1, 1, 2, 1, 1, 1],
[1, 1, 1, 2, 1, 1, 1])

>>> a = [1, 2, 3, 4, 5]
>>> np.lib.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])

>>> np.lib.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1, 0, 1, 2, 3, 4, 5, 6, 7, 8])

>>> np.lib.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])

>>> np.lib.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])

>>> np.lib.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])

>>> def padwithtens(vector, pad_width, iaxis, kwargs):
... vector[:pad_width[0]] = 10
... vector[-pad_width[1]:] = 10
... return vector

>>> a = np.arange(6)
>>> a = a.reshape((2, 3))

>>> np.lib.pad(a, 2, padwithtens)
array([[10, 10, 10, 10, 10, 10, 10],
 [10, 10, 10, 10, 10, 10, 10],
 [10, 10, 0, 1, 2, 10, 10],
 [10, 10, 3, 4, 5, 10, 10],
 [10, 10, 10, 10, 10, 10, 10],
 [10, 10, 10, 10, 10, 10, 10]])

```

**partition(a, kth, axis=-1, kind='introselect', order=None)**

Return a partitioned copy of an array.

Creates a copy of the array with its elements rearranged in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

.. versionadded:: 1.8.0

**Parameters**

a : array\_like  
Array to be sorted.

kth : int or sequence of ints  
Element index to partition by. The kth value of the element will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it.  
The order all elements in the partitions is undefined.  
If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis : int or None, optional  
Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

kind : {'introselect'}, optional  
Selection algorithm. Default is 'introselect'.

order : str or list of str, optional  
When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string. Not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the [dtype](#), to break ties.

**Returns**

partitioned\_array : ndarray  
Array of the same type and shape as `a`.

**See Also**

[ndarray.partition](#) : Method to sort an array in-place.  
[argpartition](#) : Indirect partition.  
[sort](#) : Full sorting

**Notes**

The various selection algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The available algorithms have the following properties:

kind	speed	worst case	work space	stable
------	-------	------------	------------	--------

```
=====
'introselect' 1 0(n) 0 no
=====

All the partition algorithms make temporary copies of the data when
partitioning along any but the last axis. Consequently, partitioning
along the last axis is faster and uses less space than partitioning
along any other axis.

The sort order for complex numbers is lexicographic. If both the real
and imaginary parts are non-nan then the order is determined by the
real parts except when they are equal, in which case the order is
determined by the imaginary parts.

Examples

>>> a = np.array([3, 4, 2, 1])
>>> np.partition(a, 3)
array([2, 1, 3, 4])

>>> np.partition(a, (1, 3))
array([1, 2, 3, 4])

percentile(a, q, axis=None, out=None, overwrite_input=False, interpolation='linear', keepdims=False)

Compute the qth percentile of the data along the specified axis.

Returns the qth percentile(s) of the array elements.

Parameters

a : array_like
 Input array or object that can be converted to an array.
q : float in range of [0,100] (or sequence of floats)
 Percentile to compute, which must be between 0 and 100 inclusive.
axis : {int, sequence of int, None}, optional
 Axis or axes along which the percentiles are computed. The
 default is to compute the percentile(s) along a flattened
 version of the array. A sequence of axes is supported since
 version 1.9.0.
out : ndarray, optional
 Alternative output array in which to place the result. It must
 have the same shape and buffer length as the expected output,
 but the type (of the output) will be cast if necessary.
overwrite_input : bool, optional
 If True, then allow use of memory of input array `a`
 calculations. The input array will be modified by the call to
 'percentile'. This will save memory when you do not need to
 preserve the contents of the input array. In this case you
 should not make any assumptions about the contents of the input
 'a' after this function completes -- treat it as undefined.
 Default is False. If `a` is not already an array, this parameter
 will have no effect as `a` will be converted to an array
 internally regardless of the value of this parameter.
interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}
 This optional parameter specifies the interpolation method to
 use when the desired quantile lies between two data points
 ``i < j``:
 * linear: ``i + (j - i) * fraction``, where ``fraction``
 is the fractional part of the index surrounded by ``i``
 and ``j``.
 * lower: ``i``.
 * higher: ``j``.
 * nearest: ``i`` or ``j``, whichever is nearest.
 * midpoint: ``((i + j) / 2)``.

 .. versionadded:: 1.9.0
keepdims : bool, optional
 If this is set to True, the axes which are reduced are left in
 the result as dimensions with size one. With this option, the
 result will broadcast correctly against the original array `a`.

 .. versionadded:: 1.9.0

Returns

percentile : scalar or ndarray
 If `q` is a single percentile and `axis=None`, then the result
 is a scalar. If multiple percentiles are given, first axis of
 the result corresponds to the percentiles. The other axes are
 the axes that remain after the reduction of `a`. If the input
 contains integers or floats smaller than ``float64````, the output
 data-type is ``float64``. Otherwise, the output data-type is the
 same as that of the input. If `out` is specified, that array is
 returned instead.

See Also

mean, median, nanpercentile
```

**Notes**

-----  
Given a vector ``V`` of length ``N``, the ``q``-th percentile of ``V`` is the value ``q/100`` of the way from the minimum to the maximum in in a sorted copy of ``V``. The values and distances of the two nearest neighbors as well as the `interpolation` parameter will determine the percentile if the normalized ranking does not match the location of ``q`` exactly. This function is the same as the median if ``q=50``, the same as the minimum if ``q=0`` and the same as the maximum if ``q=100``.

**Examples**

```

>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10, 7, 4],
 [3, 2, 1]])
>>> np.percentile(a, 50)
3.5
>>> np.percentile(a, 50, axis=0)
array([[6.5, 4.5, 2.5]])
>>> np.percentile(a, 50, axis=1)
array([[7., 2.]])
>>> np.percentile(a, 50, axis=1, keepdims=True)
array([[7.],
 [2.]])
```

  

```
>>> m = np.percentile(a, 50, axis=0)
>>> out = np.zeros_like(m)
>>> np.percentile(a, 50, axis=0, out=out)
array([[6.5, 4.5, 2.5]])
>>> m
array([[6.5, 4.5, 2.5]])
```

  

```
>>> b = a.copy()
>>> np.percentile(b, 50, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a == b)
```

**piecewise(x, condlist, funclist, \*args, \*\*kw)**

Evaluate a piecewise-defined function.

Given a set of conditions and corresponding functions, evaluate each function on the input data wherever its condition is true.

**Parameters**

```

x : ndarray
 The input domain.
condlist : list of bool arrays
 Each boolean array corresponds to a function in `funclist`. Wherever
 `condlist[i]` is True, `funclist[i](x)` is used as the output value.

 Each boolean array in `condlist` selects a piece of `x`,
 and should therefore be of the same shape as `x`.

 The length of `condlist` must correspond to that of `funclist`.
 If one extra function is given, i.e. if
 ``len(funclist) - len(condlist) == 1``, then that extra function
 is the default value, used wherever all conditions are false.
funclist : list of callables, f(x,*args,**kw), or scalars
 Each function is evaluated over `x` wherever its corresponding
 condition is True. It should take an array as input and give an array
 or a scalar value as output. If, instead of a callable,
 a scalar is provided then a constant function (``lambda x: scalar``) is
 assumed.
args : tuple, optional
 Any further arguments given to `piecewise` are passed to the functions
 upon execution, i.e., if called ``piecewise(..., ..., 1, 'a')``, then
 each function is called as ``f(x, 1, 'a')``.
kw : dict, optional
 Keyword arguments used in calling `piecewise` are passed to the
 functions upon execution, i.e., if called
 ``piecewise(..., ..., lambda=1)``, then each function is called as
 ``f(x, lambda=1)``.
```

**Returns**

```

out : ndarray
 The output is the same shape and type as x and is found by
 calling the functions in `funclist` on the appropriate portions of `x`,
 as defined by the boolean arrays in `condlist`. Portions not covered
 by any condition have a default value of 0.
```

**See Also**

```

choose, select, where
```

**Notes**

-----  
This is similar to choose or select, except that functions are evaluated on elements of `x` that satisfy the corresponding condition from `condlist`.

The result is::

```
|--
| funclist[0](x[condlist[0]])
out = | funclist[1](x[condlist[1]])
|...
| funclist[n2](x[condlist[n2]])
|--
```

**Examples**

-----  
Define the sigma function, which is -1 for ``x < 0`` and +1 for ``x >= 0``.

```
>>> x = np.linspace(-2.5, 2.5, 6)
>>> np.piecewise(x, [x < 0, x >= 0], [-1, 1])
array([-1., -1., -1., 1., 1., 1.])
```

Define the absolute value, which is ``-x`` for ``x < 0`` and ``x`` for ``x >= 0``.

```
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([2.5, 1.5, 0.5, 0.5, 1.5, 2.5])
```

**pkgload(\*packages, \*\*options)**

Load one or more packages into parent package top-level namespace.

This function is intended to shorten the need to import many subpackages, say of scipy, constantly with statements such as

```
import scipy.linalg, scipy.fftpack, scipy/etc...
```

Instead, you can say:

```
import scipy
scipy.pkgload('linalg','fftpack',...)
```

or

```
scipy.pkgload()
```

to load all of them in one call.

If a name which doesn't exist in scipy's namespace is given, a warning is shown.

**Parameters**

-----  
\*packages : arg-tuple  
    the names (one or more strings) of all the modules one wishes to load into the top-level namespace.  
verbose= : integer  
    verbosity level [default: -1].  
    verbose=-1 will suspend also warnings.  
force= : bool  
    when True, force reloading loaded packages [default: False].  
postpone= : bool  
    when True, don't load packages [default: False]

**place(arr, mask, vals)**

Change elements of an array based on conditional and input values.

Similar to ``np.copyto(arr, vals, where=mask)``, the difference is that `place` uses the first N elements of `vals`, where N is the [number](#) of True values in `mask`, while `copyto` uses the elements where `mask` is True.

Note that `extract` does the exact opposite of `place`.

**Parameters**

-----  
arr : ndarray  
    Array to put data into.  
mask : array\_like  
    Boolean mask array. Must have the same size as `a`.  
vals : 1-D sequence  
    Values to put into `a`. Only the first N elements are used, where N is the [number](#) of True values in `mask`. If `vals` is smaller than N it will be repeated.

**See Also**

-----  
copyto, put, take, extract

**Examples**

```

>>> arr = np.arange(6).reshape(2, 3)
>>> np.place(arr, arr>2, [44, 55])
>>> arr
array([[0, 1, 2],
 [44, 55, 44]])

pmt(rate, nper, pv, fv=0, when='end')
Compute the payment against loan principal plus interest.

Given:
* a present value, `pv` (e.g., an amount borrowed)
* a future value, `fv` (e.g., 0)
* an interest `rate` compounded once per period, of which
there are
* `nper` total
* and (optional) specification of whether payment is made
at the beginning (`when` = {'begin', 1}) or the end
(`when` = {'end', 0}) of each period

Return:
the (fixed) periodic payment.

Parameters

rate : array_like
 Rate of interest (per period)
nper : array_like
 Number of compounding periods
pv : array_like
 Present value
fv : array_like, optional
 Future value (default = 0)
when : {'begin', 1}, {'end', 0}, {string, int}
 When payments are due ('begin' (1) or 'end' (0))

Returns

out : ndarray
 Payment against loan plus interest. If all input is scalar, returns a
scalar float. If any input is array_like, returns payment for each
input element. If multiple inputs are array_like, they all must have
the same shape.

Notes

The payment is computed by solving the equation::

$$fv + \\frac{pv * (1 + rate)^{nper} + pmt * ((1 + rate)^{nper} - 1)}{rate} = 0$$

or, when ``rate == 0``:::

$$fv + pv + pmt * nper = 0$$

for ``pmt``.

Note that computing a monthly mortgage payment is only
one use for this function. For example, pmt returns the
periodic deposit one must make to achieve a specified
future balance given an initial deposit, a fixed,
periodically compounded interest rate, and the total
number of periods.

References

.. [WRW] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May).
Open Document Format for Office Applications (OpenDocument)v1.2,
Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version,
Pre-Draft 12. Organization for the Advancement of Structured Information
Standards (OASIS). Billerica, MA, USA. [ODT Document].
Available:
http://www.oasis-open.org/committees/documents.php
?wg_abbrev=office-formulaOpenDocument-formula-20090508.odt

Examples

What is the monthly payment needed to pay off a $200,000 loan in 15
years at an annual interest rate of 7.5%?

>>> np.pmt(0.075/12, 12*15, 200000)
-1854.0247200054619

In order to pay-off (i.e., have a future-value of 0) the $200,000 obtained
today, a monthly payment of $1,854.02 would be required. Note that this
example illustrates usage of `fv` having a default value of 0.
```

```
poly(seq_of_zeros)
 Find the coefficients of a polynomial with the given sequence of roots.

 Returns the coefficients of the polynomial whose leading coefficient
 is one for the given sequence of zeros (multiple roots must be included
 in the sequence as many times as their multiplicity; see Examples).
 A square matrix (or array, which will be treated as a matrix) can also
 be given, in which case the coefficients of the characteristic polynomial
 of the matrix are returned.

 Parameters

 seq_of_zeros : array_like, shape (N,) or (N, N)
 A sequence of polynomial roots, or a square array or matrix object.

 Returns

 c : ndarray
 1D array of polynomial coefficients from highest to lowest degree:
 ``c[0] * x**(N) + c[1] * x**(N-1) + ... + c[N-1] * x + c[N]``
 where c[0] always equals 1.

 Raises

 ValueError
 If input is the wrong shape (the input must be a 1-D or square
 2-D array).

 See Also

 polyval : Compute polynomial values.
 roots : Return the roots of a polynomial.
 polyfit : Least squares polynomial fit.
 poly1d : A one-dimensional polynomial class.

 Notes

 Specifying the roots of a polynomial still leaves one degree of
 freedom, typically represented by an undetermined leading
 coefficient. [1]_ In the case of this function, that coefficient -
 the first one in the returned array - is always taken as one. (If
 for some reason you have one other point, the only automatic way
 presently to leverage that information is to use ``polyfit``.)

 The characteristic polynomial, :math:`p_a(t)`, of an `n`-by-`n`
 matrix \mathbf{A} is given by

$$p_a(t) = \det(t\mathbf{I} - \mathbf{A})$$
,

 where \mathbf{I} is the `n`-by-`n` identity matrix. [2]_

 References

 .. [1] M. Sullivan and M. Sullivan, III, "Algebra and Trigonometry,
 Enhanced With Graphing Utilities," Prentice-Hall, pg. 318, 1996.

 .. [2] G. Strang, "Linear Algebra and Its Applications, 2nd Edition,"
 Academic Press, pg. 182, 1980.

 Examples

 Given a sequence of a polynomial's zeros:

$$\text{np.poly}([0, 0, 0])$$

 The line above represents $z^3 + 0z^2 + 0z + 0$.

$$\text{np.poly}([-1./2, 0, 1./2])$$

 The line above represents $z^3 - z/4$.

$$\text{np.poly((np.random.random(1.)*0, 0, np.random.random(1.)*0))}$$

 Given a square array object:

$$\text{P} = \text{np.array}([[0, 1./3], [-1./2, 0]])$$

$$\text{np.poly}(P)$$

 Or a square matrix object:

$$\text{np.poly(np.matrix}(P))$$

$$\text{array}([1. , 0. , 0.16666667])$$

```

Note how in all cases the leading coefficient is always 1.

**polyadd(a1, a2)**  
 Find the sum of two polynomials.

Returns the polynomial resulting from the sum of two input polynomials. Each input must be either a [poly1d object](#) or a 1D sequence of polynomial coefficients, from highest to lowest degree.

Parameters

a1, a2 : array\_like or [poly1d object](#)  
 Input polynomials.

Returns

out : [ndarray](#) or [poly1d object](#)  
 The sum of the inputs. If either input is a [poly1d object](#), then the output is also a [poly1d object](#). Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

See Also

[poly1d](#) : A one-dimensional polynomial class.  
[poly](#), [polyadd](#), [polyder](#), [polydiv](#), [polyfit](#), [polyint](#), [polysub](#), [polyval](#)

Examples

```
>>> np.polyadd([1, 2], [9, 5, 4])
array([9, 6, 6])
```

Using [poly1d](#) objects:

```
>>> p1 = np.poly1d([1, 2])
>>> p2 = np.poly1d([9, 5, 4])
>>> print(p1)
1 x + 2
>>> print(p2)
 2
9 x + 5 x + 4
>>> print(np.polyadd(p1, p2))
 2
9 x + 6 x + 6
```

**polyder(p, m=1)**  
 Return the derivative of the specified order of a polynomial.

Parameters

p : [poly1d](#) or sequence  
 Polynomial to differentiate.  
 A sequence is interpreted as polynomial coefficients, see [`poly1d`](#).

m : [int](#), optional  
 Order of differentiation (default: 1)

Returns

der : [poly1d](#)  
 A new polynomial representing the derivative.

See Also

[polyint](#) : Anti-derivative of a polynomial.  
[poly1d](#) : Class for one-dimensional polynomials.

Examples

The derivative of the polynomial :math:`x^3 + x^2 + x^1 + 1` is:

```
>>> p = np.poly1d([1,1,1,1])
>>> p2 = np.polyder(p)
>>> p2
poly1d([3, 2, 1])
```

which evaluates to:

```
>>> p2(2.)
17.0
```

We can verify this, approximating the derivative with  
 $``(f(x + h) - f(x))/h``$ :

```
>>> (p2(2. + 0.001) - p2(2.)) / 0.001
17.007000999997857
```

The fourth-order derivative of a 3rd-order polynomial is zero:

```
>>> np.polyder(p, 2)
poly1d([6, 2])
>>> np.polyder(p, 3)
poly1d([6])
```

```

>>> np.polyder(p, 4)
poly1d([0.])

polydiv(u, v)
 Returns the quotient and remainder of polynomial division.

 The input arrays are the coefficients (including any coefficients
 equal to zero) of the "numerator" (dividend) and "denominator"
 (divisor) polynomials, respectively.

 Parameters

 u : array_like or poly1d
 Dividend polynomial's coefficients.

 v : array_like or poly1d
 Divisor polynomial's coefficients.

 Returns

 q : ndarray
 Coefficients, including those equal to zero, of the quotient.
 r : ndarray
 Coefficients, including those equal to zero, of the remainder.

 See Also

 poly, polyadd, polyder, polydiv, polyfit, polyint, polymul, polysub,
 polyval

 Notes

 Both `u` and `v` must be 0-d or 1-d (ndim = 0 or 1), but `u.ndim` need
 not equal `v.ndim`. In other words, all four possible combinations -
 ``u.ndim = v.ndim = 0``, ``u.ndim = v.ndim = 1``,
 ``u.ndim = 1, v.ndim = 0`` , and ``u.ndim = 0, v.ndim = 1`` - work.

 Examples

 .. math:: \frac{3x^2 + 5x + 2}{2x + 1} = 1.5x + 1.75, \text{ remainder } 0.25

 >>> x = np.array([3.0, 5.0, 2.0])
 >>> y = np.array([2.0, 1.0])
 >>> np.polydiv(x, y)
 (array([1.5, 1.75]), array([0.25]))

polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
 Least squares polynomial fit.

 Fit a polynomial ``p(x) = p[0] * x**deg + ... + p[deg]`` of degree `deg`
 to points `(x, y)`. Returns a vector of coefficients `p` that minimises
 the squared error.

 Parameters

 x : array_like, shape (M,)
 x-coordinates of the M sample points ``(x[i], y[i])``.
 y : array_like, shape (M,) or (M, K)
 y-coordinates of the sample points. Several data sets of sample
 points sharing the same x-coordinates can be fitted at once by
 passing in a 2D-array that contains one dataset per column.
 deg : int
 Degree of the fitting polynomial
 rcond : float, optional
 Relative condition number of the fit. Singular values smaller than
 this relative to the largest singular value will be ignored. The
 default value is len(x)*eps, where eps is the relative precision of
 the float type, about 2e-16 in most cases.
 full : bool, optional
 Switch determining nature of return value. When it is False (the
 default) just the coefficients are returned, when True diagnostic
 information from the singular value decomposition is also returned.
 w : array_like, shape (M,), optional
 Weights to apply to the y-coordinates of the sample points. For
 gaussian uncertainties, use 1/sigma (not 1/sigma**2).
 cov : bool, optional
 Return the estimate and the covariance matrix of the estimate
 If full is True, then cov is not returned.

 Returns

 p : ndarray, shape (M,) or (M, K)
 Polynomial coefficients, highest power first. If `y` was 2-D, the
 coefficients for `k`-th data set are in ``p[:,k]``.

 residuals, rank, singular_values, rcond :
 Present only if `full` = True. Residuals of the least-squares fit,
 the effective rank of the scaled Vandermonde coefficient matrix,

```

its singular values, and the specified value of `rcond`. For more details, see `linalg.lstsq`.

`V` : `ndarray`, shape  $(M,M)$  or  $(M,M,K)$   
 Present only if `'full'` = `False` and `'cov'`=`True`. The covariance `matrix` of the polynomial coefficient estimates. The diagonal of this `matrix` are the variance estimates for each coefficient. If `y` is a 2-D array, then the covariance `matrix` for the '`k`'-th data set are in ```V[:, :, k]```

**Warns**

-----

**RankWarning**

The rank of the coefficient `matrix` in the least-squares fit is deficient. The warning is only raised if `'full'` = `False`.

The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See Also**

-----

`polyval` : Compute polynomial values.  
`linalg.lstsq` : Computes a least-squares fit.  
`scipy.interpolate.UnivariateSpline` : Computes spline fits.

**Notes**

-----

The solution minimizes the squared error

.. math ::  

$$E = \sum_{j=0}^n |p(x_j) - y_j|^2$$

in the equations:::

```
x[0]**n * p[0] + ... + x[0] * p[n-1] + p[n] = y[0]
x[1]**n * p[0] + ... + x[1] * p[n-1] + p[n] = y[1]
...
x[k]**n * p[0] + ... + x[k] * p[n-1] + p[n] = y[k]
```

The coefficient `matrix` of the coefficients `'p'` is a Vandermonde `matrix`.

`'polyfit'` issues a `'RankWarning'` when the least-squares fit is badly conditioned. This implies that the best fit is not well-defined due to numerical error. The results may be improved by lowering the polynomial degree or by replacing `'x'` by `'x' - 'x'.mean()`. The `'rcond'` parameter can also be set to a value smaller than its default, but the resulting fit may be spurious: including contributions from the small singular values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned when the degree of the polynomial is large or the interval of sample points is badly centered. The quality of the fit should always be checked in these cases. When polynomial fits are not satisfactory, splines may be a good alternative.

**References**

-----

- .. [1] Wikipedia, "Curve fitting",  
[http://en.wikipedia.org/wiki/Curve\\_fitting](http://en.wikipedia.org/wiki/Curve_fitting)
- .. [2] Wikipedia, "Polynomial interpolation",  
[http://en.wikipedia.org/wiki/Polynomial\\_interpolation](http://en.wikipedia.org/wiki/Polynomial_interpolation)

**Examples**

-----

```
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([0.08703704, -0.81349206, 1.69312169, -0.03968254])
```

It is convenient to use `'poly1d'` objects for dealing with polynomials:

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
```

```

-0.9999999999999445
>>> p30(4.5)
-0.10547061179440398

Illustration:

>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> _ = plt.plot(x, y, '.', xp, p(xp), '--', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()

polyint(p, m=1, k=None)
 Return an antiderivative (indefinite integral) of a polynomial.

 The returned order `m` antiderivative `P` of polynomial `p` satisfies

$$\frac{d^m}{dx^m} P(x) = p(x)$$
 and is defined up to `m - 1` integration constants `k`. The constants determine the low-order polynomial part

 .. math:: \frac{k_{m-1}}{m!} x^0 + \dots + \frac{k_0}{(m-1)!} x^{m-1}

 of `P` so that :math:`P^{(j)}(0) = k_{m-j-1}`.

Parameters

p : array_like or poly1d
 Polynomial to differentiate.
 A sequence is interpreted as polynomial coefficients, see `poly1d`.
m : int, optional
 Order of the antiderivative. (Default: 1)
k : list of `m` scalars or scalar, optional
 Integration constants. They are given in the order of integration: those corresponding to highest-order terms come first.

 If ``None`` (default), all constants are assumed to be zero.
 If `m = 1`, a single scalar can be given instead of a list.

See Also

polyder : derivative of a polynomial
poly1d.integ : equivalent method

Examples

The defining property of the antiderivative:

>>> p = np.poly1d([1,1,1])
>>> P = np.polyint(p)
>>> P
3 = 6 / 2!, and that the constants are given in the order of integrations. Constant of the highest-order polynomial term comes first:

>>> np.polyder(P, 2)(0)
6.0
>>> np.polyder(P, 1)(0)
5.0
>>> P(0)
3.0

polymul(a1, a2)
 Find the product of two polynomials.

 Finds the polynomial resulting from the multiplication of the two input polynomials. Each input must be either a poly1d object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

Parameters

a1, a2 : array_like or poly1d object

```

Input polynomials.

Returns

out : [ndarray](#) or [poly1d](#) object  
The polynomial resulting from the multiplication of the inputs. If either inputs is a [poly1d](#) object, then the output is also a [poly1d](#) object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

See Also

[poly1d](#) : A one-dimensional polynomial class.  
[poly](#), [polyadd](#), [polyder](#), [polydiv](#), [polyfit](#), [polyint](#), [polysub](#), [polyval](#)  
[convolve](#) : Array convolution. Same output as [polymul](#), but has parameter for overlap mode.

Examples

```
>>> np.polymul([1, 2, 3], [9, 5, 1])
array([9, 23, 38, 17, 3])
```

Using [poly1d](#) objects:

```
>>> p1 = np.poly1d([1, 2, 3])
>>> p2 = np.poly1d([9, 5, 1])
>>> print(p1)
 2
1 x + 2 x + 3
>>> print(p2)
 2
9 x + 5 x + 1
>>> print(np.polymul(p1, p2))
 4 3 2
9 x + 23 x + 38 x + 17 x + 3
```

### **polysub(a1, a2)**

Difference (subtraction) of two polynomials.

Given two polynomials 'a1' and 'a2', returns ``a1 - a2``.  
'a1' and 'a2' can be either array\_like sequences of the polynomials' coefficients (including coefficients equal to zero), or [poly1d](#) objects.

Parameters

a1, a2 : array\_like or [poly1d](#)  
Minuend and subtrahend polynomials, respectively.

Returns

out : [ndarray](#) or [poly1d](#)  
Array or [poly1d](#) object of the difference polynomial's coefficients.

See Also

[polyval](#), [polydiv](#), [polymul](#), [polyadd](#)

Examples

```
.. math:: (2 x^2 + 10 x - 2) - (3 x^2 + 10 x - 4) = (-x^2 + 2)

>>> np.polysub([2, 10, -2], [3, 10, -4])
array([-1, 0, 2])
```

### **polyval(p, x)**

Evaluate a polynomial at specific values.

If 'p' is of length N, this function returns the value:

$$``p[0]*x^{N-1} + p[1]*x^{N-2} + \dots + p[N-2]*x + p[N-1]``$$

If 'x' is a sequence, then 'p(x)' is returned for each element of 'x'.  
If 'x' is another polynomial then the composite polynomial 'p(x(t))' is returned.

Parameters

p : array\_like or [poly1d](#) object  
1D array of polynomial coefficients (including coefficients equal to zero) from highest degree to the constant term, or an instance of [poly1d](#).  
x : array\_like or [poly1d](#) object  
A [number](#), an array of numbers, or an instance of [poly1d](#), at which to evaluate 'p'.

Returns

**values** : `ndarray` or `poly1d`  
 If `x` is a `poly1d` instance, the result is the composition of the two polynomials, i.e., `x` is "substituted" in `p`, and the simplified result is returned. In addition, the type of `x` - array\_like or `poly1d` - governs the type of the output: `x` array\_like => `values` array\_like, `x` a `poly1d object` => `values` is also.

See Also

`poly1d`: A polynomial class.

Notes

Horner's scheme [1]\_ is used to evaluate the polynomial. Even so, for polynomials of high degree the values may be inaccurate due to rounding errors. Use carefully.

References

.. [1] I. N. Bronshtein, K. A. Semendyayev, and K. A. Hirsch (Eng. trans. Ed.), \*Handbook of Mathematics\*, New York, Van Nostrand Reinhold Co., 1985, pg. 720.

Examples

```
>>> np.polyval([3,0,1], 5) # 3 * 5**2 + 0 * 5**1 + 1
76
>>> np.polyval([3,0,1], np.poly1d(5))
poly1d([76.])
>>> np.polyval(np.poly1d([3,0,1]), 5)
76
>>> np.polyval(np.poly1d([3,0,1]), np.poly1d(5))
poly1d([76.])
```

## ppmt(rate, per, nper, pv, fv=0.0, when='end')

Compute the payment against loan principal.

Parameters

`rate` : array\_like  
 Rate of interest (per period)  
`per` : array\_like, `int`  
 Amount paid against the loan changes. The `per` is the period of interest.  
`nper` : array\_like  
 Number of compounding periods  
`pv` : array\_like  
 Present value  
`fv` : array\_like, optional  
 Future value  
`when` : {`'begin'`, 1}, {`'end'`, 0}, {string, `int`}  
 When payments are due ('begin' (1) or 'end' (0))

See Also

`pmt`, `pv`, `ipmt`

## prod(a, axis=None, dtype=None, out=None, keepdims=False)

Return the product of array elements over a given axis.

Parameters

`a` : array\_like  
 Input data.  
`axis` : None or `int` or tuple of ints, optional  
 Axis or axes along which a product is performed. The default, axis=None, will calculate the product of all the elements in the input array. If axis is negative it counts from the last to the first axis.  
 .. versionadded:: 1.7.0  
 If axis is a tuple of ints, a product is performed on all of the axes specified in the tuple instead of a `single` axis or all the axes as before.  
`dtype` : `dtype`, optional  
 The type of the returned array, as well as of the accumulator in which the elements are multiplied. The `dtype` of `a` is used by default unless `a` has an `integer dtype` of less precision than the default platform `integer`. In that case, if `a` is signed then the platform `integer` is used while if `a` is unsigned then an unsigned `integer` of the same precision as the platform `integer` is used.  
`out` : `ndarray`, optional  
 Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.  
`keepdims` : bool, optional  
 If this is set to True, the axes which are reduced are left in the

```
result as dimensions with size one. With this option, the result
will broadcast correctly against the input array.

>Returns

product_along_axis : ndarray, see `dtype` parameter above.
 An array shaped as ``a`` but with the specified axis removed.
 Returns a reference to `out` if specified.

See Also

ndarray.prod : equivalent method
numpy.doc.ufuncs : Section "Output arguments"

Notes

Arithmetic is modular when using integer types, and no error is
raised on overflow. That means that, on a 32-bit platform:

>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x) #random
16

The product of an empty array is the neutral element 1:

>>> np.prod([])
1.0

Examples

By default, calculate the product of all elements:

>>> np.prod([1.,2.])
2.0

Even when the input array is two-dimensional:

>>> np.prod([[1.,2.],[3.,4.]])
24.0

But we can also specify the axis over which to multiply:

>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([2., 12.])

If the type of ``x`` is unsigned, then the output type is
the unsigned platform integer:

>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True

If ``x`` is of a signed integer type, then the output type
is the default platform integer:

>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == np.int
True

product(a, axis=None, dtype=None, out=None, keepdims=False)
Return the product of array elements over a given axis.

See Also

prod : equivalent function; see for details.

promote_types(...)
promote_types(type1, type2)

Returns the data type with the smallest size and smallest scalar
kind to which both ``type1`` and ``type2`` may be safely cast.
The returned data type is always in native byte order.

This function is symmetric and associative.

Parameters

type1 : dtype or dtype specifier
 First data type.
type2 : dtype or dtype specifier
 Second data type.

>Returns

out : dtype
 The promoted data type.

Notes
```

```

.. versionadded:: 1.6.0

Starting in NumPy 1.9, promote_types function now returns a valid string
length when given an integer or float dtype as one argument and a string
dtype as another argument. Previously it always returned the input string
dtype, even if it wasn't long enough to store the max integer/float value
converted to a string.

See Also

result_type, dtype, can_cast

Examples

>>> np.promote_types('f4', 'f8')
dtype('float64')

>>> np.promote_types('i8', 'f4')
dtype('float64')

>>> np.promote_types('>i8', '<c8')
dtype('complex128')

>>> np.promote_types('i4', 'S8')
dtype('S11')

ptp(a, axis=None, out=None)
Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

Parameters

a : array_like
 Input values.
axis : int, optional
 Axis along which to find the peaks. By default, flatten the
 array.
out : array_like
 Alternative output array in which to place the result. It must
 have the same shape and buffer length as the expected output,
 but the type of the output values will be cast if necessary.

Returns

ptp : ndarray
 A new array holding the result, unless `out` was
 specified, in which case a reference to `out` is returned.

Examples

>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
 [2, 3]])

>>> np.ptp(x, axis=0)
array([2, 2])

>>> np.ptp(x, axis=1)
array([1, 1])

put(a, ind, v, mode='raise')
Replaces specified elements of an array with given values.

The indexing works on the flattened target array. `put` is roughly
equivalent to:

:::
 a.flat[ind] = v

Parameters

a : ndarray
 Target array.
ind : array_like
 Target indices, interpreted as integers.
v : array_like
 Values to place in `a` at target indices. If `v` is shorter than
 `ind` it will be repeated as necessary.
mode : {'raise', 'wrap', 'clip'}, optional
 Specifies how out-of-bounds indices will behave.

 * 'raise' -- raise an error (default)
 * 'wrap' -- wrap around
 * 'clip' -- clip to the range
```

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

See Also

-----

`putmask`, `place`

Examples

-----

```
>>> a = np.arange(5)
>>> np.put(a, [0, 2], [-44, -55])
>>> a
array([-44, 1, -55, 3, 4])

>>> a = np.arange(5)
>>> np.put(a, 22, -5, mode='clip')
>>> a
array([0, 1, 2, 3, -5])
```

**putmask(...)**

`putmask(a, mask, values)`

Changes elements of an array based on conditional and input values.

Sets ```a.flat[n] = values[n]``` for each `n` where ```mask.flat[n]==True```.

If `values` is not the same size as `a` and `mask` then it will repeat. This gives behavior different from ```a[mask] = values```.

Parameters

-----

`a` : `array_like`  
Target array.

`mask` : `array_like`  
Boolean `mask` array. It has to be the same shape as `a`.

`values` : `array_like`  
Values to put into `a` where `mask` is True. If `values` is smaller than `a` it will be repeated.

See Also

-----

`place`, `put`, `take`, `copyto`

Examples

-----

```
>>> x = np.arange(6).reshape(2, 3)
>>> np.putmask(x, x>2, x**2)
>>> x
array([[0, 1, 2],
 [9, 16, 25]])
```

If `values` is smaller than `a` it is repeated:

```
>>> x = np.arange(5)
>>> np.putmask(x, x>1, [-33, -44])
>>> x
array([0, 1, -33, -44, -33])
```

**pv(rate, nper, pmt, fv=0.0, when='end')**

Compute the present value.

Given:

- \* a future value, `fv`
- \* an interest `rate` compounded once per period, of which there are
- \* `nper` total
- \* a (fixed) payment, `pmt`, paid either
- \* at the beginning (`when` = {'begin', 1}) or the end (`when` = {'end', 0}) of each period

Return:

the value now

Parameters

-----

`rate` : `array_like`  
Rate of interest (per period)

`nper` : `array_like`  
Number of compounding periods

`pmt` : `array_like`  
Payment

`fv` : `array_like`, optional  
Future value

`when` : {{'begin', 1}, {'end', 0}}, {string, `int`}, optional  
When payments are due ('begin' (1) or 'end' (0))

**Returns**  
 -----  
**out : ndarray, float**  
 Present value of a series of payments or investments.

**Notes**  
 -----  
 The present value is computed by solving the equation:::

```
fv +
pv*(1 + rate)**nper +
pmt*(1 + rate**when)/rate*((1 + rate)**nper - 1) = 0
```

or, when ``rate = 0``:::

$$fv + pv + pmt * nper = 0$$

for `pv`, which is then returned.

**References**  
 -----  
... [WRW] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May). Open Document Format for Office Applications (OpenDocument)v1.2, Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version, Pre-Draft 12. Organization for the Advancement of Structured Information Standards (OASIS). Billerica, MA, USA. [ODT Document]. Available:  
[http://www.oasis-open.org/committees/documents.php?wg\\_abbrev=office-formula](http://www.oasis-open.org/committees/documents.php?wg_abbrev=office-formula)  
OpenDocument-formula-20090508.odt

**Examples**  
 -----  
 What is the present value (e.g., the initial investment) of an investment that needs to total \$15692.93 after 10 years of saving \$100 every month? Assume the interest rate is 5% (annually) compounded monthly.

```
>>> np.pv(0.05/12, 10*12, -100, 15692.93)
-100.00067131625819
```

By convention, the negative sign represents cash flow out (i.e., money not available today). Thus, to end up with \$15,692.93 in 10 years saving \$100 a month at 5% annual interest, one's initial deposit should also be \$100.

If any input is array\_like, ``pv`` returns an array of equal shape. Let's compare different interest rates in the example above:

```
>>> a = np.array((0.05, 0.04, 0.03))/12
>>> np.pv(a, 10*12, -100, 15692.93)
array([-100.00067132, -649.26771385, -1273.78633713])
```

So, to end up with the same \$15692.93 under the same \$100 per month "savings plan," for annual interest rates of 4% and 3%, one would need initial investments of \$649.27 and \$1273.79, respectively.

**rank(a)**  
 Return the number of dimensions of an array.  
 If `a` is not already an array, a conversion is attempted. Scalars are zero dimensional.  
 ... note::  
 This function is deprecated in NumPy 1.9 to avoid confusion with `numpy.linalg.matrix\_rank`. The ``ndim`` attribute or function should be used instead.

**Parameters**  
 -----  
**a : array\_like**  
 Array whose number of dimensions is desired. If `a` is not an array, a conversion is attempted.

**Returns**  
 -----  
**number\_of\_dimensions : int**  
 The number of dimensions in the array.

**See Also**  
 -----  
**ndim** : equivalent function  
**ndarray.ndim** : equivalent property  
**shape** : dimensions of array  
**ndarray.shape** : dimensions of array

**Notes**  
 -----  
 In the old Numeric package, `rank` was the term used for the number of dimensions, but in Numpy `ndim` is used instead.

```
Examples

>>> np.rank([1,2,3])
1
>>> np.rank(np.array([[1,2,3],[4,5,6]]))
2
>>> np.rank(1)
0

rate(nper, pmt, pv, fv, when='end', guess=0.1, tol=1e-06, maxiter=100)
Compute the rate of interest per period.

Parameters

nper : array_like
 Number of compounding periods
pmt : array_like
 Payment
pv : array_like
 Present value
fv : array_like
 Future value
when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
 When payments are due ('begin' (1) or 'end' (0))
guess : float, optional
 Starting guess for solving the rate of interest
tol : float, optional
 Required tolerance for the solution
maxiter : int, optional
 Maximum iterations in finding the solution

Notes

The rate of interest is computed by iteratively solving the
(non-linear) equation::

 fv + pv*(1+rate)**nper + pmt*(1+rate*when)/rate * ((1+rate)**nper - 1) = 0

for ``rate``.

References

Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May). Open Document
Format for Office Applications (OpenDocument)v1.2, Part 2: Recalculated
Formula (OpenFormula) Format - Annotated Version, Pre-Draft 12.
Organization for the Advancement of Structured Information Standards
(OASIS). Billerica, MA, USA. [ODT Document]. Available:
http://www.oasis-open.org/committees/documents.php?wg_abbrev=office-formula
OpenDocument-formula-20090508.odt

ravel(a, order='C')
Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is
made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input
array. (for example, a masked array will be returned for a masked array
input)

Parameters

a : array_like
 Input array. The elements in `a` are read in the order specified by
 `order`, and packed as a 1-D array.
order : {'C', 'F', 'A', 'K'}, optional

 The elements of `a` are read using this index order. 'C' means
 to index the elements in row-major, C-style order,
 with the last axis index changing fastest, back to the first
 axis index changing slowest. 'F' means to index the elements
 in column-major, Fortran-style order, with the
 first index changing fastest, and the last index changing
 slowest. Note that the 'C' and 'F' options take no account of
 the memory layout of the underlying array, and only refer to
 the order of axis indexing. 'A' means to read the elements in
 Fortran-like index order if `a` is Fortran *contiguous* in
 memory, C-like order otherwise. 'K' means to read the
 elements in the order they occur in memory, except for
 reversing the data when strides are negative. By default, 'C'
 index order is used.

Returns

y : array_like
 If `a` is a matrix, y is a 1-D ndarray, otherwise y is an array of
 the same subtype as `a`. The shape of the returned array is
```

```

``(a.size,)``. Matrices are special cased for backward
compatibility.

See Also

ndarray.flat : 1-D iterator over an array.
ndarray.flatten : 1-D array copy of the elements of an array
 in row-major order.
ndarray.reshape : Change the shape of an array without changing its data.

Notes

In row-major, C-style order, in two dimensions, the row index
varies the slowest, and the column index the quickest. This can
be generalized to multiple dimensions, where row-major order
implies that the index along the first axis varies slowest, and
the index along the last quickest. The opposite holds for
column-major, Fortran-style index ordering.

When a view is desired in as many cases as possible, ``arr.reshape(-1)``
may be preferable.

Examples

It is equivalent to ``reshape(-1, order=order)``.

>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(np.ravel(x))
[1 2 3 4 5 6]

>>> print(x.reshape(-1))
[1 2 3 4 5 6]

>>> print(np.ravel(x, order='F'))
[1 4 2 5 3 6]

When ``order`` is 'A', it will preserve the array's 'C' or 'F' ordering:

>>> print(np.ravel(x.T))
[1 4 2 5 3 6]
>>> print(np.ravel(x.T, order='A'))
[1 2 3 4 5 6]

When ``order`` is 'K', it will preserve orderings that are neither 'C'
nor 'F', but won't reverse axes:

>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])

>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[0, 2, 4],
 [1, 3, 5]],
 [[6, 8, 10],
 [7, 9, 11]]])
>>> a.ravel(order='C')
array([0, 2, 4, 1, 3, 5, 6, 8, 10, 7, 9, 11])
>>> a.ravel(order='K')
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

ravel_multi_index(...)
ravel_multi_index(multi_index, dims, mode='raise', order='C')

Converts a tuple of index arrays into an array of flat
indices, applying boundary modes to the multi-index.

Parameters

multi_index : tuple of array_like
 A tuple of integer arrays, one array for each dimension.
dims : tuple of ints
 The shape of array into which the indices from ``multi_index`` apply.
mode : {'raise', 'wrap', 'clip'}, optional
 Specifies how out-of-bounds indices are handled. Can specify
 either one mode or a tuple of modes, one mode per index.
 * 'raise' -- raise an error (default)
 * 'wrap' -- wrap around
 * 'clip' -- clip to the range

 In 'clip' mode, a negative index which would normally
 wrap will clip to 0 instead.
order : {'C', 'F'}, optional
 Determines whether the multi-index should be viewed as
 indexing in row-major (C-style) or column-major
 (Fortran-style) order.

```

Returns  
-----  
`raveled_indices : ndarray`  
An array of indices into the flattened version of an array  
of dimensions ``dims``.

See Also  
-----  
`unravel_index`

Notes  
-----  
.. versionadded:: 1.6.0

Examples  
-----  
`>>> arr = np.array([[3,6,6],[4,5,1]])`  
`>>> np.ravel_multi_index(arr, (7,6))`  
`array([22, 41, 37])`  
`>>> np.ravel_multi_index(arr, (7,6), order='F')`  
`array([31, 41, 13])`  
`>>> np.ravel_multi_index(arr, (4,6), mode='clip')`  
`array([22, 23, 19])`  
`>>> np.ravel_multi_index(arr, (4,4), mode=('clip','wrap'))`  
`array([12, 13, 13])`

`>>> np.ravel_multi_index((3,1,4,1), (6,7,8,9))`  
1621

**real(val)**  
Return the real part of the elements of the array.

Parameters  
-----  
`val : array_like`  
Input array.

Returns  
-----  
`out : ndarray`  
Output array. If `val` is real, the type of `val` is used for the output. If `val` has `complex` elements, the returned type is `float`.

See Also  
-----  
`real_if_close`, `imag`, `angle`

Examples  
-----  
`>>> a = np.array([1+2j, 3+4j, 5+6j])`  
`>>> a.real`  
`array([ 1., 3., 5.])`  
`>>> a.real = 9`  
`>>> a`  
`array([ 9.+2.j, 9.+4.j, 9.+6.j])`  
`>>> a.real = np.array([9, 8, 7])`  
`>>> a`  
`array([ 9.+2.j, 8.+4.j, 7.+6.j])`

**real\_if\_close(a, tol=100)**  
If `complex` input returns a real array if `complex` parts are close to zero.  
"Close to zero" is defined as `tol` \* (machine epsilon of the type for `a`).

Parameters  
-----  
`a : array_like`  
Input array.  
`tol : float`  
Tolerance in machine epsilon for the `complex` part of the elements in the array.

Returns  
-----  
`out : ndarray`  
If `a` is real, the type of `a` is used for the output. If `a` has `complex` elements, the returned type is `float`.

See Also  
-----  
`real`, `imag`, `angle`

Notes  
-----  
Machine epsilon varies from machine to machine and between data types but Python floats on most platforms have a machine epsilon equal to

```
2.2204460492503131e-16. You can use 'np.finfo(np.float).eps' to print
out the machine epsilon for floats.

Examples

>>> np.finfo(np.float).eps
2.2204460492503131e-16

>>> np.real_if_close([2.1 + 4e-14j], tol=1000)
array([2.1])
>>> np.real_if_close([2.1 + 4e-13j], tol=1000)
array([2.1 +4.0000000e-13j])

recfromcsv(fname, **kwargs)
Load ASCII data stored in a comma-separated file.

The returned array is a record array (if ``usemask=False``, see
`recarray`) or a masked record array (if ``usemask=True``,
see `ma.mrecords.MaskedRecords`).

Parameters

fname, kwargs : For a description of input parameters, see `genfromtxt`.

See Also

numpy.genfromtxt : generic function to load ASCII data.

Notes

By default, ``dtype`` is None, which means that the data-type of the output
array will be determined from the data.

recfromtxt(fname, **kwargs)
Load ASCII data from a file and return it in a record array.

If ``usemask=False`` a standard recarray is returned,
if ``usemask=True`` a MaskedRecords array is returned.

Parameters

fname, kwargs : For a description of input parameters, see `genfromtxt`.

See Also

numpy.genfromtxt : generic function

Notes

By default, ``dtype`` is None, which means that the data-type of the output
array will be determined from the data.

repeat(a, repeats, axis=None)
Repeat elements of an array.

Parameters

a : array_like
 Input array.
repeats : int or array of ints
 The number of repetitions for each element. `repeats` is broadcasted
 to fit the shape of the given axis.
axis : int, optional
 The axis along which to repeat values. By default, use the
 flattened input array, and return a flat output array.

Returns

repeated_array : ndarray
 Output array which has the same shape as `a`, except along
 the given axis.

See Also

tile : Tile an array.

Examples

>>> x = np.array([[1,2],[3,4]])
>>> np.repeat(x, 2)
array([1, 1, 2, 2, 3, 3, 4, 4])
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 1, 2, 2],
 [3, 3, 3, 4, 4]])
>>> np.repeat(x, [1, 2], axis=0)
array([[1, 2],
 [3, 4],
 [3, 4]])
```

```
require(a, dtype=None, requirements=None)
 Return an ndarray of the provided type that satisfies requirements.

 This function is useful to be sure that an array with the correct flags
 is returned for passing to compiled code (perhaps through ctypes).

Parameters

a : array_like
 The object to be converted to a type-and-requirement-satisfying array.
dtype : data-type
 The required data-type. If None preserve the current dtype. If your
 application requires the data to be in native byteorder, include
 a byteorder specification as a part of the dtype specification.
requirements : str or list of str
 The requirements list can be any of the following

 * 'F_CONTIGUOUS' ('F') - ensure a Fortran-contiguous array
 * 'C_CONTIGUOUS' ('C') - ensure a C-contiguous array
 * 'ALIGNED' ('A') - ensure a data-type aligned array
 * 'WRITEABLE' ('W') - ensure a writable array
 * 'OWNDATA' ('O') - ensure an array that owns its own data
 * 'ENSUREARRAY', ('E') - ensure a base array, instead of a subclass

See Also

asarray : Convert input to an ndarray.
asanyarray : Convert to an ndarray, but pass through ndarray subclasses.
ascontiguousarray : Convert input to a contiguous array.
asfortranarray : Convert input to an ndarray with column-major
 memory order.
ndarray.flags : Information about the memory layout of the array.

Notes

The returned array will be guaranteed to have the listed requirements
by making a copy if needed.

Examples

>>> x = np.arange(6).reshape(2,3)
>>> x.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

>>> y = np.require(x, dtype=np.float32, requirements=['A', 'O', 'W', 'F'])
>>> y.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

reshape(a, newshape, order='C')
 Gives a new shape to an array without changing its data.

Parameters

a : array_like
 Array to be reshaped.
newshape : int or tuple of ints
 The new shape should be compatible with the original shape. If
 an integer, then the result will be a 1-D array of that length.
 One shape dimension can be -1. In this case, the value is inferred
 from the length of the array and remaining dimensions.
order : {'C', 'F', 'A'}, optional
 Read the elements of `a` using this index order, and place the elements
 into the reshaped array using this index order. 'C' means to
 read / write the elements using C-like index order, with the last axis
 index changing fastest, back to the first axis index changing slowest.
 'F' means to read / write the elements using Fortran-like index order,
 with the first index changing fastest, and the last index changing
 slowest.
 Note that the 'C' and 'F' options take no account of the memory layout
 of the underlying array, and only refer to the order of indexing. 'A'
 means to read / write the elements in Fortran-like index order if `a`
 is Fortran *contiguous* in memory, C-like order otherwise.

Returns

reshaped_array : ndarray
 This will be a new view object if possible; otherwise, it will
 be a copy. Note there is no guarantee of the *memory layout* (C- or
 Fortran- contiguous) of the returned array.
```

## See Also

[ndarray.reshape](#) : Equivalent method.

## Notes

It is not always possible to change the shape of an array without copying the data. If you want an error to be raised if the data is copied, you should assign the new shape to the shape attribute of the array::

```
>>> a = np.zeros((10, 2))
A transpose make the array non-contiguous
>>> b = a.T
Taking a view makes it possible to modify the shape without modifying
the initial object.
>>> c = b.view()
>>> c.shape = (20)
AttributeError: incompatible shape for a non-contiguous array
```

The `order` keyword gives the index ordering both for \*fetching\* the values from `a`, and then \*placing\* the values into the output array.

For example, let's say you have an array:

```
>>> a = np.arange(6).reshape((3, 2))
>>> a
array([[0, 1],
 [2, 3],
 [4, 5]])
```

You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```
>>> np.reshape(a, (2, 3)) # C-like index ordering
array([[0, 1, 2],
 [3, 4, 5]])
>>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape
array([[0, 1, 2],
 [3, 4, 5]])
>>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering
array([[0, 4, 3],
 [2, 1, 5]])
>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
 [2, 1, 5]])
```

## Examples

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])

>>> np.reshape(a, (3,-1)) # the unspecified value is inferred to be 2
array([[1, 2],
 [3, 4],
 [5, 6]])
```

**resize(a, new\_shape)**

Return a new array with the specified shape.

If the new array is larger than the original array, then the new array is filled with repeated copies of `a`. Note that this behavior is different from `a.resize(new_shape)` which fills with zeros instead of repeated copies of `a`.

## Parameters

`a` : `array_like`  
Array to be resized.

`new_shape` : `int` or tuple of `int`  
Shape of resized array.

## Returns

`reshaped_array` : `ndarray`  
The new array is formed from the data in the old array, repeated if necessary to fill out the required `number` of elements. The data are repeated in the order that they are stored in memory.

## See Also

[ndarray.resize](#) : resize an array in-place.

## Examples

```

>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(2,3))
array([[0, 1, 2],
 [3, 0, 1]])
>>> np.resize(a,(1,4))
array([[0, 1, 2, 3]])
>>> np.resize(a,(2,4))
array([[0, 1, 2, 3],
 [0, 1, 2, 3]])

restoredot()
Restore `dot`, `vdot`, and `innerproduct` to the default non-BLAS
implementations.

Typically, the user will only need to call this when troubleshooting
and installation problem, reproducing the conditions of a build without
an accelerated BLAS, or when being very careful about benchmarking
linear algebra operations.

.. note:: Deprecated in Numpy 1.10
 The cblas functions have been integrated into the multarray
 module and restoredot now longer does anything. It will be
 removed in Numpy 1.11.0.

See Also

alteredot : `restoredot` undoes the effects of `alteredot`.

result_type(...)
result_type(*arrays_and_dtypes)

Returns the type that results from applying the NumPy
type promotion rules to the arguments.

Type promotion in NumPy works similarly to the rules in languages
like C++, with some slight differences. When both scalars and
arrays are used, the array's type takes precedence and the actual value
of the scalar is taken into account.

For example, calculating 3*a, where a is an array of 32-bit floats,
intuitively should result in a 32-bit float output. If the 3 is a
32-bit integer, the NumPy rules indicate it can't convert losslessly
into a 32-bit float, so a 64-bit float should be the result type.
By examining the value of the constant, '3', we see that it fits in
an 8-bit integer, which can be cast losslessly into the 32-bit float.

Parameters

arrays_and_dtypes : list of arrays and dtypes
 The operands of some operation whose result type is needed.

Returns

out : dtype
 The result type.

See also

dtype, promote_types, min_scalar_type, can_cast

Notes

.. versionadded:: 1.6.0

The specific algorithm used is as follows.

Categories are determined by first checking which of boolean,
integer (int/uint), or floating point (float/complex) the maximum
kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars
is higher than the maximum category of the arrays,
the data types are combined with :func:`promote_types`
to produce the return value.

Otherwise, 'min_scalar_type' is called on each array, and
the resulting data types are all combined with :func:`promote_types`
to produce the return value.

The set of int values is not a subset of the uint values for types
with the same number of bits, something not reflected in
:func:`min_scalar_type`, but handled as a special case in `result_type`.

Examples

>>> np.result_type(3, np.arange(7, dtype='i1'))
dtype('int8')
```

```
>>> np.result_type('i4', 'c8')
dtype('complex128')

>>> np.result_type(3.0, -2)
dtype('float64')

roll(a, shift, axis=None)
 Roll array elements along a given axis.

 Elements that roll beyond the last position are re-introduced at
 the first.

 Parameters

 a : array_like
 Input array.
 shift : int
 The number of places by which elements are shifted.
 axis : int, optional
 The axis along which elements are shifted. By default, the array
 is flattened before shifting, after which the original
 shape is restored.

 Returns

 res : ndarray
 Output array, with the same shape as `a`.

 See Also

 rollaxis : Roll the specified axis backwards, until it lies in a
 given position.

 Examples

 >>> x = np.arange(10)
 >>> np.roll(x, 2)
 array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])

 >>> x2 = np.reshape(x, (2,5))
 >>> x2
 array([[0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9]])
 >>> np.roll(x2, 1)
 array([[9, 0, 1, 2, 3],
 [4, 5, 6, 7, 8]])
 >>> np.roll(x2, 1, axis=0)
 array([[5, 6, 7, 8, 9],
 [0, 1, 2, 3, 4]])
 >>> np.roll(x2, 1, axis=1)
 array([[4, 0, 1, 2, 3],
 [9, 5, 6, 7, 8]])

rollaxis(a, axis, start=0)
 Roll the specified axis backwards, until it lies in a given position.

 Parameters

 a : ndarray
 Input array.
 axis : int
 The axis to roll backwards. The positions of the other axes do not
 change relative to one another.
 start : int, optional
 The axis is rolled until it lies before this position. The default,
 0, results in a "complete" roll.

 Returns

 res : ndarray
 For Numpy >= 1.10 a view of `a` is always returned. For earlier
 Numpy versions a view of `a` is returned only if the order of the
 axes is changed, otherwise the input array is returned.

 See Also

 moveaxis : Move array axes to new positions.
 roll : Roll the elements of an array by a number of positions along a
 given axis.

 Examples

 >>> a = np.ones((3,4,5,6))
 >>> np.rollaxis(a, 3, 1).shape
 (3, 6, 4, 5)
 >>> np.rollaxis(a, 2).shape
 (5, 3, 4, 6)
```

```
>>> np.rollaxis(a, 1, 4).shape
(3, 5, 6, 4)

roots(p)
Return the roots of a polynomial with coefficients given in p.

The values in the rank-1 array `p` are coefficients of a polynomial.
If the length of `p` is n+1 then the polynomial is described by::
 p[0] * x**n + p[1] * x**(n-1) + ... + p[n-1]*x + p[n]

Parameters

p : array_like
 Rank-1 array of polynomial coefficients.

Returns

out : ndarray
 An array containing the complex roots of the polynomial.

Raises

ValueError
 When `p` cannot be converted to a rank-1 array.

See also

poly : Find the coefficients of a polynomial with a given sequence
 of roots.
polyval : Compute polynomial values.
polyfit : Least squares polynomial fit.
poly1d : A one-dimensional polynomial class.

Notes

The algorithm relies on computing the eigenvalues of the
companion matrix [1]_.

References

.. [1] R. A. Horn & C. R. Johnson, *Matrix Analysis*. Cambridge, UK:
 Cambridge University Press, 1999, pp. 146-7.

Examples

>>> coeff = [3.2, 2, 1]
>>> np.roots(coeff)
array([-0.3125+0.46351241j, -0.3125-0.46351241j])

rot90(m, k=1)
Rotate an array by 90 degrees in the counter-clockwise direction.

The first two dimensions are rotated; therefore, the array must be at
least 2-D.

Parameters

m : array_like
 Array of two or more dimensions.
k : integer
 Number of times the array is rotated by 90 degrees.

Returns

y : ndarray
 Rotated array.

See Also

fliplr : Flip an array horizontally.
flipud : Flip an array vertically.

Examples

>>> m = np.array([[1,2],[3,4]], int)
>>> m
array([[1, 2],
 [3, 4]])
>>> np.rot90(m)
array([[2, 4],
 [1, 3]])
>>> np.rot90(m, 2)
array([[4, 3],
 [2, 1]])

round_(a, decimals=0, out=None)
Round an array to the given number of decimals.
```

```
Refer to `around` for full documentation.

See Also

around : equivalent function

row_stack = vstack(tup)
Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single
array. Rebuild arrays divided by `vsplit`.

Parameters

tup : sequence of ndarrays
 Tuple containing arrays to be stacked. The arrays must have the same
 shape along all but the first axis.

Returns

stacked : ndarray
 The array formed by stacking the given arrays.

See Also

stack : Join a sequence of arrays along a new axis.
hstack : Stack arrays in sequence horizontally (column wise).
dstack : Stack arrays in sequence depth wise (along third dimension).
concatenate : Join a sequence of arrays along an existing axis.
vsplit : Split array into a list of multiple sub-arrays vertically.

Notes

Equivalent to ``np.concatenate(tup, axis=0)`` if `tup` contains arrays that
are at least 2-dimensional.

Examples

>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
 [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1,
 [2,
 [3,
 [2,
 [3,
 [4]]]]]

safe_eval(source)
Protected string evaluation.

Evaluate a string containing a Python literal expression without
allowing the execution of arbitrary non-literal code.

Parameters

source : str
 The string to evaluate.

Returns

obj : object
 The result of evaluating `source`.

Raises

SyntaxError
 If the code has invalid Python syntax, or if it contains
 non-literal code.

Examples

>>> np.safe_eval('1')
1
>>> np.safe_eval('[1, 2, 3]')
[1, 2, 3]
>>> np.safe_eval('{"foo": ("bar", 10.0)}')
{'foo': ('bar', 10.0)}

>>> np.safe_eval('import os')
Traceback (most recent call last):
```

```
...
SyntaxError: invalid syntax
>>> np.safe_eval('open("/home/user/.ssh/id_dsa").read()')
Traceback (most recent call last):
...
SyntaxError: Unsupported source construct: compiler.ast.CallFunc

save(file, arr, allow_pickle=True, fix_imports=True)
Save an array to a binary file in NumPy ``.npy`` format.

Parameters

file : file or str
 File or filename to which the data is saved. If file is a file-object, then the filename is unchanged. If file is a string, a ``.npy`` extension will be appended to the file name if it does not already have one.
allow_pickle : bool, optional
 Allow saving object arrays using Python pickles. Reasons for disallowing pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations, for example if the stored objects require libraries that are not available, and not all pickled data is compatible between Python 2 and Python 3).
 Default: True
fix_imports : bool, optional
 Only useful in forcing objects in object arrays on Python 3 to be pickled in a Python 2 compatible way. If `fix_imports` is True, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.
arr : array_like
 Array data to be saved.

See Also

savez : Save several arrays into a ``.npz`` archive
savetxt, load

Notes

For a description of the ``.npy`` format, see the module docstring of `numpy.lib.format` or the Numpy Enhancement Proposal http://docs.scipy.org/doc/numpy/neps/npy-format.html

Examples

>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()

>>> x = np.arange(10)
>>> np.save(outfile, x)

>>> outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> np.load(outfile)
array[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]

savetxt(fname, X, fmt='%1.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ')
Save an array to a text file.

Parameters

fname : filename or file handle
 If the filename ends in ``.gz``, the file is automatically saved in compressed gzip format. `loadtxt` understands gzipped files transparently.
X : array_like
 Data to be saved to a text file.
fmt : str or sequence of strs, optional
 A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. 'Iteration %d -- %10.5f', in which case `delimiter` is ignored. For complex `X`, the legal options for `fmt` are:
 a) a single specifier, `fmt='%.4e'`, resulting in numbers formatted like ``(%s+%sj)` % (fmt, fmt)`
 b) a full string specifying every real and imaginary part, e.g.
 ``%.4e %+4j %.4e %+4j %.4e %+4j`` for 3 columns
 c) a list of specifiers, one per column - in this case, the real and imaginary part must have separate specifiers,
 e.g. ``[%'.3e + %.3ej', '%(.15e%+.15ej)']`` for 2 columns
delimiter : str, optional
 String or character separating columns.
newline : str, optional
 String or character separating lines.

.. versionadded:: 1.5.0
header : str, optional
 String that will be written at the beginning of the file.
```

```
.. versionadded:: 1.7.0
footer : str, optional
 String that will be written at the end of the file.

.. versionadded:: 1.7.0
comments : str, optional
 String that will be prepended to the ``header`` and ``footer`` strings,
 to mark them as comments. Default: '# ', as expected by e.g.
 ``numpy.loadtxt``.

.. versionadded:: 1.7.0

See Also

save : Save an array to a binary file in NumPy ``.npy`` format
savez : Save several arrays into an uncompressed ``.npz`` archive
savez_compressed : Save several arrays into a compressed ``.npz`` archive

Notes

Further explanation of the `fmt` parameter
(``%[flag]width [.precision]specifier``):

flags:
``-`` : left justify
``+`` : Forces to precede result with + or -.
``0`` : Left pad the number with zeros instead of space (see width).

width:
Minimum number of characters to be printed. The value is not truncated
if it has more characters.

precision:
- For integer specifiers (eg. ``d,i,o,x``), the minimum number of
 digits.
- For ``e``, ``E`` and ``f`` specifiers, the number of digits to print
 after the decimal point.
- For ``g`` and ``G``, the maximum number of significant digits.
- For ``s``, the maximum number of characters.

specifiers:
``c`` : character
``d`` or ``i`` : signed decimal integer
``e`` or ``E`` : scientific notation with ``e`` or ``E``.
``f`` : decimal floating point
``g,G`` : use the shorter of ``e,E`` or ``f``.
``o`` : signed octal
``s`` : string of characters
``u`` : unsigned decimal integer
``x,X`` : unsigned hexadecimal integer

This explanation of ``fmt`` is not complete, for an exhaustive
specification see [1]_.

References

.. [1] `Format Specification Mini-Language
 <http://docs.python.org/library/string.html#format-specification-mini-language>`, Python Documentation.

Examples

>>> x = y = z = np.arange(0.0,5.0,1.0)
>>> np.savetxt('test.out', x, delimiter=',') # X is an array
>>> np.savetxt('test.out', (x,y,z)) # x,y,z equal sized 1D arrays
>>> np.savetxt('test.out', x, fmt='%1.4e') # use exponential notation

savez(file, *args, **kwds)
Save several arrays into a single file in uncompressed ``.npz`` format.

If arguments are passed in with no keywords, the corresponding variable
names, in the ``.npz`` file, are 'arr_0', 'arr_1', etc. If keyword
arguments are given, the corresponding variable names, in the ``.npz``
file will match the keyword names.

Parameters

```

```
file : str or file
 Either the file name (string) or an open file (file-like object)
 where the data will be saved. If file is a string, the ``.npz``
 extension will be appended to the file name if it is not already there.
args : Arguments, optional
 Arrays to save to the file. Since it is not possible for Python to
 know the names of the arrays outside `savez`, the arrays will be saved
 with names "arr_0", "arr_1", and so on. These arguments can be any
 expression.
kwds : Keyword arguments, optional
 Arrays to save to the file. Arrays will be saved in the file with the
 keyword names.

Returns

None

See Also

save : Save a single array to a binary file in NumPy format.
savetxt : Save an array to a file as plain text.
savez_compressed : Save several arrays into a compressed ``.npz`` archive

Notes

The ``.npz`` file format is a zipped archive of files named after the
variables they contain. The archive is not compressed and each file
in the archive contains one variable in ``.npy`` format. For a
description of the ``.npy`` format, see `numpy.lib.format` or the
Numpy Enhancement Proposal
http://docs.scipy.org/doc/numpy/neps/npy-format.html

When opening the saved ``.npz`` file with `load` a `NpzFile` object is
returned. This is a dictionary-like object which can be queried for
its list of arrays (with the ``.files`` attribute), and for the arrays
themselves.

Examples

>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> x = np.arange(10)
>>> y = np.sin(x)

Using `savez` with *args, the arrays are saved with default names.

>>> np.savez(outfile, x, y)
>>> outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> npzfile = np.load(outfile)
>>> npzfile.files
['arr_1', 'arr_0']
>>> npzfile['arr_0']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Using `savez` with **kwds, the arrays are saved with the keyword names.

>>> outfile = TemporaryFile()
>>> np.savez(outfile, x=x, y=y)
>>> outfile.seek(0)
>>> npzfile = np.load(outfile)
>>> npzfile.files
['y', 'x']
>>> npzfile['x']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

savez_compressed(file, *args, **kwds)
Save several arrays into a single file in compressed ``.npz`` format.

If keyword arguments are given, then filenames are taken from the keywords.
If arguments are passed in with no keywords, then stored file names are
arr_0, arr_1, etc.

Parameters

file : str
 File name of ``.npz`` file.
args : Arguments
 Function arguments.
kwds : Keyword arguments
 Keywords.

See Also

numpy.savez : Save several arrays into an uncompressed ``.npz`` file format
numpy.load : Load the files created by savez_compressed.
```

**sctype2char**(sctype)  
Return the string representation of a scalar [dtype](#).

```
Parameters

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

Returns

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

Raises

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

See Also

obj2sctype, issctype, issubscotype, mintypecode

Examples

>>> for sctype in [np.int32, np.float, np.complex, np.string, np.ndarray]:
... print(np.sctype2char(sctype))
l
d
D
S
0

>>> x = np.array([1., 2-1.j])
>>> np.sctype2char(x)
'D'
>>> np.sctype2char(list)
'0'

searchsorted(a, v, side='left', sorter=None)
Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array `a` such that, if the
corresponding elements in `v` were inserted before the indices, the
order of `a` would be preserved.

Parameters

a : 1-D array_like
 Input array. If `sorter` is None, then it must be sorted in
 ascending order, otherwise `sorter` must be an array of indices
 that sort it.
v : array_like
 Values to insert into `a`.
side : {'left', 'right'}, optional
 If 'left', the index of the first suitable location found is given.
 If 'right', return the last such index. If there is no suitable
 index, return either 0 or N (where N is the length of `a`).
sorter : 1-D array_like, optional
 Optional array of integer indices that sort array a into ascending
 order. They are typically the result of argsort.

.. versionadded:: 1.7.0

Returns

indices : array of ints
 Array of insertion points with the same shape as `v`.

See Also

sort : Return a sorted copy of an array.
histogram : Produce histogram from 1-D data.

Notes

Binary search is used to find the required insertion points.

As of Numpy 1.4.0 `searchsorted` works with real/complex arrays containing
`nan` values. The enhanced sort order is documented in `sort`.

Examples

>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])
```

```
select(condlist, choicelist, default=0)
 Return an array drawn from elements in choicelist, depending on conditions.

 Parameters

 condlist : list of bool ndarrays
 The list of conditions which determine from which array in `choicelist`
 the output elements are taken. When multiple conditions are satisfied,
 the first one encountered in `condlist` is used.
 choicelist : list of ndarrays
 The list of arrays from which the output elements are taken. It has
 to be of the same length as `condlist`.
 default : scalar, optional
 The element inserted in `output` when all conditions evaluate to False.

 Returns

 output : ndarray
 The output at position m is the m-th element of the array in
 `choicelist` where the m-th element of the corresponding array in
 `condlist` is True.

 See Also

 where : Return elements from one of two arrays depending on condition.
 take, choose, compress, diag, diagonal

 Examples

 >>> x = np.arange(10)
 >>> condlist = [x<3, x>5]
 >>> choicelist = [x, x**2]
 >>> np.select(condlist, choicelist)
 array([0, 1, 2, 0, 0, 0, 36, 49, 64, 81])

set_numeric_ops(...)
 set_numeric_ops(op1=func1, op2=func2, ...)

 Set numerical operators for array objects.

 Parameters

 op1, op2, ... : callable
 Each ``op = func`` pair describes an operator to be replaced.
 For example, ``add = lambda x, y: np.add(x, y) % 5`` would replace
 addition by modulus 5 addition.

 Returns

 saved_ops : list of callables
 A list of all operators, stored before making replacements.

 Notes

 .. WARNING::
 Use with care! Incorrect usage may lead to memory errors.

 A function replacing an operator cannot make use of that operator.
 For example, when replacing add, you may not use ``+``. Instead,
 directly call ufuncs.

 Examples

 >>> def add_mod5(x, y):
 ... return np.add(x, y) % 5
 ...
 >>> old_funcs = np.set_numeric_ops(add=add_mod5)

 >>> x = np.arange(12).reshape((3, 4))
 >>> x + x
 array([[0, 2, 4, 1],
 [3, 0, 2, 4],
 [1, 3, 0, 2]])

 >>> ignore = np.set_numeric_ops(**old_funcs) # restore operators

set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=None, suppress=None,
nanstr=None, infstr=None, formatter=None)
 Set printing options.

 These options determine the way floating point numbers, arrays and
 other NumPy objects are displayed.

 Parameters

 precision : int, optional
 Number of digits of precision for floating point output (default 8).
 threshold : int, optional
```

```

Total number of array elements which trigger summarization
rather than full repr (default 1000).

edgeitems : int, optional
 Number of array items in summary at beginning and end of
 each dimension (default 3).

linewidth : int, optional
 The number of characters per line for the purpose of inserting
 line breaks (default 75).

suppress : bool, optional
 Whether or not suppress printing of small floating point values
 using scientific notation (default False).

nanstr : str, optional
 String representation of floating point not-a-number (default nan).

infstr : str, optional
 String representation of floating point infinity (default inf).

formatter : dict of callables, optional
 If not None, the keys should indicate the type(s) that the respective
 formatting function applies to. Callables should return a string.
 Types that are not specified (by their corresponding keys) are handled
 by the default formatters. Individual types for which a formatter
 can be set are::

 - 'bool'
 - 'int'
 - 'timedelta' : a `numpy.timedelta64``
 - 'datetime' : a `numpy.datetime64``
 - 'float'
 - 'longfloat' : 128-bit floats
 - 'complexfloat'
 - 'longcomplexfloat' : composed of two 128-bit floats
 - 'numpy_`str`' : types `numpy.string`_` and `numpy.unicode`_
 - 'str' : all other strings

Other keys that can be used to set a group of types at once are::

 - 'all' : sets all types
 - 'int_kind' : sets 'int'
 - 'float_kind' : sets 'float' and 'longfloat'
 - 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
 - 'str_kind' : sets 'str' and 'numpystr'

See Also

get_printoptions, set_string_function, array2string

Notes

`formatter` is always reset with a call to set_printoptions.

Examples

Floating point precision can be set:

>>> np.set_printoptions(precision=4)
>>> print(np.array([1.123456789]))
[1.1235]

Long arrays can be summarised:

>>> np.set_printoptions(threshold=5)
>>> print(np.arange(10))
[0 1 2 ..., 7 8 9]

Small results can be suppressed:

>>> eps = np.finfo(float).eps
>>> x = np.arange(4.)
>>> x**2 - (x + eps)**2
array([-4.9304e-32, -4.4409e-16, 0.0000e+00, 0.0000e+00])
>>> np.set_printoptions(suppress=True)
>>> x**2 - (x + eps)**2
array([-0., -0., 0., 0.])

A custom formatter can be used to display array elements as desired:

>>> np.set_printoptions(formatter={'all':lambda x: 'int: '+str(-x)})
>>> x = np.arange(3)
>>> x
array([int: 0, int: -1, int: -2])
>>> np.set_printoptions() # formatter gets reset
>>> x
array([0, 1, 2])

To put back the default options, you can use:

>>> np.set_printoptions(edgeitems=3,infstr='inf',
... linewidth=75, nanstr='nan', precision=8,
... suppress=False, threshold=1000, formatter=None)

```

```
set_string_function(f, repr=True)
 Set a Python function to be used when pretty printing arrays.

 Parameters

 f : function or None
 Function to be used to pretty print arrays. The function should expect
 a single array argument and return a string of the representation of
 the array. If None, the function is reset to the default NumPy function
 to print arrays.
 repr : bool, optional
 If True (default), the function for pretty printing (`__repr__`)
 is set, if False the function that returns the default string
 representation (`__str__`) is set.

 See Also

 set_printoptions, get_printoptions

 Examples

 >>> def pprint(arr):
... return 'HA! - What are you going to do now?'
...
>>> np.set_string_function(pprint)
>>> a = np.arange(10)
>>> a
HA! - What are you going to do now?
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]

We can reset the function to the default:
>>> np.set_string_function(None)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

`repr` affects either pretty printing or normal string representation.
Note that `__repr__` is still affected by setting `__str__`
because the width of each array element in the returned string becomes
equal to the length of the result of `__str__()`.

>>> x = np.arange(4)
>>> np.set_string_function(lambda x:'random', repr=False)
>>> x.__str__()
'random'
>>> x.__repr__()
'array([0, 1, 2, 3])'

setbufsize(size)
 Set the size of the buffer used in ufuncs.

 Parameters

 size : int
 Size of buffer.

setdiff1d(ar1, ar2, assume_unique=False)
 Find the set difference of two arrays.

 Return the sorted, unique values in `ar1` that are not in `ar2`.

 Parameters

 ar1 : array_like
 Input array.
 ar2 : array_like
 Input comparison array.
 assume_unique : bool
 If True, the input arrays are both assumed to be unique, which
 can speed up the calculation. Default is False.

 Returns

 setdiff1d : ndarray
 Sorted 1D array of values in `ar1` that are not in `ar2`.

 See Also

 numpy.lib.arraysetops : Module with a number of other functions for
 performing set operations on arrays.

 Examples

 >>> a = np.array([1, 2, 3, 2, 4, 1])
 >>> b = np.array([3, 4, 5, 6])
 >>> np.setdiff1d(a, b)
 array([1, 2])
```

```
seterr(all=None, divide=None, over=None, under=None, invalid=None)
 Set how floating-point errors are handled.

 Note that operations on integer scalar types (such as 'int16') are
 handled like floating point, and are affected by these settings.

 Parameters

 all : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
 Set treatment for all types of floating point errors at once:
 - ignore: Take no action when the exception occurs.
 - warn: Print a RuntimeWarning (via the Python `warnings` module).
 - raise: Raise a FloatingPointError.
 - call: Call a function specified using the 'seterrcall' function.
 - print: Print a warning directly to 'stdout'.
 - log: Record error in a Log object specified by 'seterrcall'.

 The default is not to change the current behavior.

 divide : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
 Treatment for division by zero.

 over : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
 Treatment for floating-point overflow.

 under : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
 Treatment for floating-point underflow.

 invalid : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
 Treatment for invalid floating-point operation.

 Returns

 old_settings : dict
 Dictionary containing the old settings.

 See also

 seterrcall : Set a callback function for the 'call' mode.
 geterr, geterrcall, errstate

 Notes

 The floating-point exceptions are defined in the IEEE 754 standard [1]:
 - Division by zero: infinite result obtained from finite numbers.
 - Overflow: result too large to be expressed.
 - Underflow: result so close to zero that some precision
 was lost.
 - Invalid operation: result is not an expressible number, typically
 indicates that a NaN was produced.

 .. [1] http://en.wikipedia.org/wiki/IEEE_754

 Examples

 >>> old_settings = np.seterr(all='ignore') #seterr to known value
 >>> np.seterr(over='raise')
 {'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore',
 'under': 'ignore'}
 >>> np.seterr(**old_settings) # reset to default
 {'over': 'raise', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}

 >>> np.int16(32000) * np.int16(3)
 30464
 >>> old_settings = np.seterr(all='warn', over='raise')
 >>> np.int16(32000) * np.int16(3)
 Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 FloatingPointError: overflow encountered in short_scalars

 >>> old_settings = np.seterr(all='print')
 >>> np.geterr()
 {'over': 'print', 'divide': 'print', 'invalid': 'print', 'under': 'print'}
 >>> np.int16(32000) * np.int16(3)
 Warning: overflow encountered in short_scalars
 30464

seterrcall(func)
 Set the floating-point error callback function or log object.

 There are two ways to capture floating-point error messages. The first
 is to set the error-handler to 'call', using 'seterr'. Then, set
 the function to call using this function.

 The second is to set the error-handler to 'log', using 'seterr'.
 Floating-point errors then trigger a call to the 'write' method of
 the provided object.

 Parameters

 func : callable f(err, flag) or object with write method
```

Function to call upon [floating-point errors](#) ('call'-mode) or [object](#) whose 'write' method is used to log such message ('log'-mode).

The call function takes two arguments. The first is a string describing the type of error (such as "divide by zero", "overflow", "underflow", or "invalid value"), and the second is the status flag. The flag is a [byte](#), whose four least-significant bits indicate the type of error, one of "divide", "over", "under", "invalid":

```
[0 0 0 0 divide over under invalid]
```

In other words, ``flags = divide + 2\*over + 4\*under + 8\*invalid``.

If an [object](#) is provided, its write method should take one argument, a string.

Returns

```

```

h : callable, log instance or None  
The old error handler.

See Also

```

```

[seterr](#), [geterr](#), [geterrcall](#)

Examples

```

```

Callback upon error:

```
>>> def err_handler(type, flag):
... print("Floating point error (%s), with flag %s" % (type, flag))
...

>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')

>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([Inf, Inf, Inf])

>>> np.seterrcall(saved_handler)
<function err_handler at 0x...>
>>> np.seterr(**save_err)
{'over': 'call', 'divide': 'call', 'invalid': 'call', 'under': 'call'}
```

Log error message:

```
>>> class Log(object):
... def write(self, msg):
... print("LOG: %s" % msg)
...

>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')

>>> np.array([1, 2, 3]) / 0.0
LOG: Warning: divide by zero encountered in divide
<BLANKLINE>
array([Inf, Inf, Inf])

>>> np.seterrcall(saved_handler)
<__main__.Log object at 0x...>
>>> np.seterr(**save_err)
{'over': 'log', 'divide': 'log', 'invalid': 'log', 'under': 'log'}
```

**seterrobj(...)**  
[seterrobj](#)(errorobj)

Set the [object](#) that defines [floating-point error handling](#).

The error [object](#) contains all information that defines the error handling behavior in Numpy. `seterrobj` is used internally by the other functions that set error handling behavior (`seterr`, `seterrcall`).

Parameters

```

```

errorobj : list  
The error [object](#), a list containing three elements:  
[internal numpy buffer size, error mask, error callback function].

The error mask is a [single integer](#) that holds the treatment information on all four [floating point errors](#). The information for each error type is contained in three bits of the [integer](#). If we print it in base 8, we can see what treatment is set for "invalid", "under", "over", and "divide" (in that order). The printed string can be interpreted with

```
* 0 : 'ignore'
* 1 : 'warn'
```

```
* 2 : 'raise'
* 3 : 'call'
* 4 : 'print'
* 5 : 'log'

See Also

geterrobj, seterr, geterr, seterrcall, geterrcall
getbufsize, setbufsize

Notes

For complete documentation of the types of floating-point exceptions and
treatment options, see `seterr`.

Examples

>>> old_errobj = np.geterrobj() # first get the defaults
>>> old_errobj
[10000, 0, None]

>>> def err_handler(type, flag):
... print("Floating point error (%s), with flag %s" % (type, flag))
...
>>> new_errobj = [20000, 12, err_handler]
>>> np.seterrobj(new_errobj)
>>> np.base_repr(12, 8) # int for divide=4 ('print') and over=1 ('warn')
'14'
>>> np.geterr()
{'over': 'warn', 'divide': 'print', 'invalid': 'ignore', 'under': 'ignore'}
>>> np.geterrcall() is err_handler
True

setxor1d(ar1, ar2, assume_unique=False)
Find the set exclusive-or of two arrays.

Return the sorted, unique values that are in only one (not both) of the
input arrays.

Parameters

ar1, ar2 : array_like
 Input arrays.
assume_unique : bool
 If True, the input arrays are both assumed to be unique, which
 can speed up the calculation. Default is False.

Returns

setxor1d : ndarray
 Sorted 1D array of unique values that are in only one of the input
 arrays.

Examples

>>> a = np.array([1, 2, 3, 2, 4])
>>> b = np.array([2, 3, 5, 7, 5])
>>> np.setxor1d(a,b)
array([1, 4, 5, 7])

shape(a)
Return the shape of an array.

Parameters

a : array_like
 Input array.

Returns

shape : tuple of ints
 The elements of the shape tuple give the lengths of the
 corresponding array dimensions.

See Also

alen
ndarray.shape : Equivalent array method.

Examples

>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
```

```
()
=>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
=>> np.shape(a)
(2,)
>>> a.shape
(2,)

shares_memory(...)
shares_memory(a, b, max_work=None)

Determine if two arrays share memory

Parameters

a, b : ndarray
 Input arrays
max_work : int, optional
 Effort to spend on solving the overlap problem (maximum number
 of candidate solutions to consider). The following special
 values are recognized:
 max_work=MAY_SHARE_EXACT (default)
 -The problem is solved exactly. In this case, the function returns
 True only if there is an element shared between the arrays.
 max_work=MAY_SHARE_BOUNDS
 Only the memory bounds of a and b are checked.

Raises

numpy.TooHardError
 Exceeded max_work.

Returns

out : bool

See Also

may_share_memory

Examples

=>> np.may_share_memory(np.array([1,2]), np.array([5,8,9]))
False

show_config = show()

sinc(x)
Return the sinc function.

The sinc function is :math:`\sin(\pi x)/(\pi x)`.

Parameters

x : ndarray
 Array (possibly multi-dimensional) of values for which to calculate ``sinc(x)``.

Returns

out : ndarray
 ``sinc(x)``, which has the same shape as the input.

Notes

``sinc(0)`` is the limit value 1.

The name sinc is short for "sine cardinal" or "sinus cardinalis".

The sinc function is used in various signal processing applications,
including in anti-aliasing, in the construction of a Lanczos resampling
filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal
interpolation kernel is proportional to the sinc function.

References

.. [1] Weisstein, Eric W. "Sinc Function." From MathWorld--A Wolfram Web
 Resource. http://mathworld.wolfram.com/SincFunction.html
.. [2] Wikipedia, "Sinc function",
 http://en.wikipedia.org/wiki/Sinc_function

Examples

=>>> x = np.linspace(-4, 4, 41)
=>>> np.sinc(x)
```

```
array([-3.89804309e-17, -4.92362781e-02, -8.40918587e-02,
 -8.90384387e-02, -5.84680802e-02, 3.89804309e-17,
 6.68206631e-02, 1.16434881e-01, 1.26137788e-01,
 8.50444803e-02, -3.89804309e-17, -1.03943254e-01,
 -1.89206682e-01, -2.16236208e-01, -1.55914881e-01,
 3.89804309e-17, 2.33872321e-01, 5.04551152e-01,
 7.56826729e-01, 9.35489284e-01, 1.00000000e+00,
 9.35489284e-01, 7.56826729e-01, 5.04551152e-01,
 2.33872321e-01, 3.89804309e-17, -1.55914881e-01,
 -2.16236208e-01, -1.89206682e-01, -1.03943254e-01,
 -3.89804309e-17, 8.50444803e-02, 1.26137788e-01,
 1.16434881e-01, 6.68206631e-02, 3.89804309e-17,
 -5.84680802e-02, -8.90384387e-02, -8.40918587e-02,
 -4.92362781e-02, -3.89804309e-17])
```

```
>>> plt.plot(x, np.sinc(x))
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Sinc Function")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("X")
<matplotlib.text.Text object at 0x...>
>>> plt.show()
```

It works in 2-D as well:

```
>>> x = np.linspace(-4, 4, 401)
>>> xx = np.outer(x, x)
>>> plt.imshow(np.sinc(xx))
<matplotlib.image.AxesImage object at 0x...>
```

## size(a, axis=None)

Return the number of elements along a given axis.

### Parameters

```

a : array_like
 Input data.
axis : int, optional
 Axis along which the elements are counted. By default, give
 the total number of elements.
```

### Returns

```

element_count : int
 Number of elements along the specified axis.
```

### See Also

```

shape : dimensions of array
ndarray.shape : dimensions of array
ndarray.size : number of elements in array
```

### Examples

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.size(a)
6
>>> np.size(a,1)
3
>>> np.size(a,0)
2
```

## sometrue(a, axis=None, out=None, keepdims=False)

Check whether some values are true.

Refer to `any` for full documentation.

### See Also

```

any : equivalent function
```

## sort(a, axis=-1, kind='quicksort', order=None)

Return a sorted copy of an array.

### Parameters

```

a : array_like
 Array to be sorted.
axis : int or None, optional
 Axis along which to sort. If None, the array is flattened before
 sorting. The default is -1, which sorts along the last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
 Sorting algorithm. Default is 'quicksort'.
order : str or list of str, optional
 When 'a' is an array with fields defined, this argument specifies
 which fields to compare first, second, etc. A single field can
```

```

be specified as a string, and not all fields need be specified,
but unspecified fields will still be used, in the order in which
they come up in the dtype, to break ties.

>Returns

sorted_array : ndarray
 Array of the same type and shape as `a`.

See Also

ndarray.sort : Method to sort an array in-place.
argsort : Indirect sort.
lexsort : Indirect stable sort on multiple keys.
searchsorted : Find elements in a sorted array.
partition : Partial sort.

Notes

The various sorting algorithms are characterized by their average speed,
worst case performance, work space size, and whether they are stable. A
stable sort keeps items with the same key in the same relative
order. The three available algorithms have the following
properties:

=====
 kind speed worst case work space stable
===== ===== ===== ===== =====
'quicksort' 1 O(n^2) 0 no
'mergesort' 2 O(n*log(n)) ~n/2 yes
'heapsort' 3 O(n*log(n)) 0 no
===== ===== ===== ===== =====

All the sort algorithms make temporary copies of the data when
sorting along any but the last axis. Consequently, sorting along
the last axis is faster and uses less space than sorting along
any other axis.

The sort order for complex numbers is lexicographic. If both the real
and imaginary parts are non-nan then the order is determined by the
real parts except when they are equal, in which case the order is
determined by the imaginary parts.

Previous to numpy 1.4.0 sorting real and complex arrays containing nan
values led to undefined behaviour. In numpy versions >= 1.4.0 nan
values are sorted to the end. The extended sort order is:

* Real: [R, nan]
* Complex: [R + Rj, R + nanj, nan + Rj, nan + nanj]

where R is a non-nan real value. Complex values with the same nan
placements are sorted according to the non-nan part if it exists.
Non-nan values are sorted as before.

Examples

>>> a = np.array([[1,4],[3,1]])
>>> np.sort(a) # sort along the last axis
array([[1, 4],
 [3, 1]])
>>> np.sort(a, axis=None) # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0) # sort along the first axis
array([[1, 1],
 [3, 4]])

Use the `order` keyword to specify a field to use when sorting a
structured array:

>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
... ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype) # create a structured array
>>> np.sort(a, order='height') # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
 ('Lancelot', 1.8999999999999999, 38)],
 dtype=\[\('name', '|S10'\), \('height', '<f8'\), \('age', '<i4'\)\]\)

Sort by age, then height if ages are equal:

>>> np.sort(a, order=['age', 'height']) # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
 ('Arthur', 1.8, 41)],
 dtype=\[\('name', '|S10'\), \('height', '<f8'\), \('age', '<i4'\)\]\)
```

**sort\_complex(a)**

Sort a [complex](#) array using the real part first, then the imaginary part.

Parameters

```

a : array_like
 Input array

Returns

out : complex ndarray
 Always returns a sorted complex array.

Examples

>>> np.sort_complex([5, 3, 6, 2, 1])
array([1.+0.j, 2.+0.j, 3.+0.j, 5.+0.j, 6.+0.j])

>>> np.sort_complex([1 + 2j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])
array([1.+2.j, 2.-1.j, 3.-3.j, 3.-2.j, 3.+5.j])

source(object, output=<open file '<stdout>', mode 'w')

Print or write to a file the source code for a Numpy object.

The source code is only returned for objects written in Python. Many

functions and classes are defined in C and will therefore not return

useful information.

Parameters

object : numpy object
 Input object. This can be any object (function, class, module,

 ...).

output : file object, optional
 If `output` not supplied then source code is printed to screen

 (sys.stdout). File object must be created with either write 'w' or

 append 'a' modes.

See Also

lookfor, info

Examples

>>> np.source(np.interp) #doctest: +SKIP
In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
def interp(x, xp, fp, left=None, right=None):
 """.... (full docstring printed)"""
 if isinstance(x, (float, int, number)):
 return compiled_interp([x], xp, fp, left, right).item()
 else:
 return compiled_interp(x, xp, fp, left, right)

The source code is only returned for objects written in Python.

>>> np.source(np.array) #doctest: +SKIP
Not available for this object.
```

**split**(ary, indices\_or\_sections, axis=0)  
Split an array into multiple sub-arrays.

Parameters

-----

**ary** : ndarray  
Array to be divided into sub-arrays.  
**indices\_or\_sections** : int or 1-D array  
If `indices\_or\_sections` is an **integer**, N, the array will be divided  
into N equal arrays along `axis`. If such a split is not possible,  
an error is raised.  
If `indices\_or\_sections` is a 1-D array of sorted integers, the entries  
indicate where along `axis` the array is split. For example,  
`[2, 3]` would, for `axis=0`, result in  

- ary[:2]
- ary[2:3]
- ary[3:]

If an index exceeds the dimension of the array along `axis`,  
an empty sub-array is returned correspondingly.  
**axis** : int, optional  
The axis along which to split, default is 0.

Returns

-----

**sub-arrays** : list of ndarrays  
A list of sub-arrays.

Raises

-----

**ValueError**  
If `indices\_or\_sections` is given as an **integer**, but

```

 a split does not result in equal division.

See Also

array_split : Split an array into multiple sub-arrays of equal or
 near-equal size. Does not raise an exception if
 an equal division cannot be made.
hsplit : Split array into multiple sub-arrays horizontally (column-wise).
vsplit : Split array into multiple sub-arrays vertically (row wise).
dsplit : Split array into multiple sub-arrays along the 3rd axis (depth).
concatenate : Join a sequence of arrays along an existing axis.
stack : Join a sequence of arrays along a new axis.
hstack : Stack arrays in sequence horizontally (column wise).
vstack : Stack arrays in sequence vertically (row wise).
dstack : Stack arrays in sequence depth wise (along third dimension).

Examples

>>> x = np.arange(9.0)
>>> np.split(x, 3)
[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7., 8.])]

>>> x = np.arange(8.0)
>>> np.split(x, [3, 5, 6, 10])
[array([0., 1., 2.]),
 array([3., 4.]),
 array([5.]),
 array([6., 7.]),
 array([], dtype=float64)]

squeeze(a, axis=None)
Remove single-dimensional entries from the shape of an array.

Parameters

a : array_like
 Input data.
axis : None or int or tuple of ints, optional
 .. versionadded:: 1.7.0

 Selects a subset of the single-dimensional entries in the
 shape. If an axis is selected with shape entry greater than
 one, an error is raised.

Returns

squeezed : ndarray
 The input array, but with all or a subset of the
 dimensions of length 1 removed. This is always `a` itself
 or a view into `a`.

Examples

>>> x = np.array([[0], [1], [2]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=(2,)).shape
(1, 3)

stack(arrays, axis=0)
Join a sequence of arrays along a new axis.

The `axis` parameter specifies the index of the new axis in the dimensions
of the result. For example, if ``axis=0`` it will be the first dimension
and if ``axis=-1`` it will be the last dimension.

.. versionadded:: 1.10.0

Parameters

arrays : sequence of array_like
 Each array must have the same shape.
axis : int, optional
 The axis in the result array along which the input arrays are stacked.

Returns

stacked : ndarray
 The stacked array has one more dimension than the input arrays.

See Also

concatenate : Join a sequence of arrays along an existing axis.
split : Split array into a list of multiple sub-arrays of equal size.

Examples

```

```

>>> arrays = [np.random.randn(3, 4) for _ in range(10)]
>>> np.stack(arrays, axis=0).shape
(10, 3, 4)

>>> np.stack(arrays, axis=1).shape
(3, 10, 4)

>>> np.stack(arrays, axis=2).shape
(3, 4, 10)

>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.stack((a, b))
array([[1, 2, 3],
 [2, 3, 4]])

>>> np.stack((a, b), axis=-1)
array([[1, 2],
 [2, 3],
 [3, 4]])

std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
 Compute the standard deviation along the specified axis.

 Returns the standard deviation, a measure of the spread of a distribution,
 of the array elements. The standard deviation is computed for the
 flattened array by default, otherwise over the specified axis.

 Parameters

 a : array_like
 Calculate the standard deviation of these values.
 axis : None or int or tuple of ints, optional
 Axis or axes along which the standard deviation is computed. The
 default is to compute the standard deviation of the flattened array.

 .. versionadded:: 1.7.0

 If this is a tuple of ints, a standard deviation is performed over
 multiple axes, instead of a single axis or all the axes as before.

 dtype : dtype, optional
 Type to use in computing the standard deviation. For arrays of
 integer type the default is float64, for arrays of float types it is
 the same as the array type.

 out : ndarray, optional
 Alternative output array in which to place the result. It must have
 the same shape as the expected output but the type (of the calculated
 values) will be cast if necessary.

 ddof : int, optional
 Means Delta Degrees of Freedom. The divisor used in calculations
 is ``N - ddof``, where ``N`` represents the number of elements.
 By default `ddof` is zero.

 keepdims : bool, optional
 If this is set to True, the axes which are reduced are left
 in the result as dimensions with size one. With this option,
 the result will broadcast correctly against the original `arr`.

 Returns

 standard deviation : ndarray, see dtype parameter above.
 If `out` is None, return a new array containing the standard deviation,
 otherwise return a reference to the output array.

 See Also

 var, mean, nanmean, nanstd, nanvar
 numpy.doc.ufuncs : Section "Output arguments"

 Notes

 The standard deviation is the square root of the average of the squared
 deviations from the mean, i.e., ``std = sqrt(mean(abs(x - x.mean\(\)**2\)\)\)``.

 The average squared deviation is normally calculated as
 ``x.sum\(\) / N``, where ``N = len\(x\)``. If, however, `ddof` is specified,
 the divisor ``N - ddof`` is used instead. In standard statistical
 practice, ``ddof=1`` provides an unbiased estimator of the variance
 of the infinite population. ``ddof=0`` provides a maximum likelihood
 estimate of the variance for normally distributed variables. The
 standard deviation computed in this function is the square root of
 the estimated variance, so even with ``ddof=1``, it will not be an
 unbiased estimate of the standard deviation per se.

 Note that, for complex numbers, `std` takes the absolute
 value before squaring, so that the result is always real and nonnegative.

 For floating-point input, the *std* is computed using the same
 precision the input has. Depending on the input data, this can cause
```

the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-accuracy accumulator using the `'dtype'` keyword can alleviate this issue.

#### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([1., 1.])
>>> np.std(a, axis=1)
array([0.5, 0.5])
```

In `single` precision, `std()` can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.std(a)
0.45000005
```

Computing the standard deviation in `float64` is more accurate:

```
>>> np.std(a, dtype=np.float64)
0.4499999925494177
```

**sum(a, axis=None, dtype=None, out=None, keepdims=False)**  
Sum of array elements over a given axis.

#### Parameters

-----

`a` : `array_like`

Elements to sum.

`axis` : `None` or `int` or tuple of ints, optional

Axis or axes along which a sum is performed. The default, `axis=None`, will sum all of the elements of the input array. If `axis` is negative it counts from the last to the first axis.

.. versionadded:: 1.7.0

If `axis` is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a `single` axis or all the axes as before.

`dtype` : `dtype`, optional

The type of the returned array and of the accumulator in which the elements are summed. The `dtype` of `'a'` is used by default unless `'a'` has an `integer dtype` of less precision than the default platform `integer`. In that case, if `'a'` is signed then the platform `integer` is used while if `'a'` is unsigned then an unsigned `integer` of the same precision as the platform `integer` is used.

`out` : `ndarray`, optional

Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

`keepdims` : `bool`, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will `broadcast` correctly against the input array.

#### Returns

-----

`sum_along_axis` : `ndarray`

An array with the same shape as `'a'`, with the specified axis removed. If `'a'` is a 0-d array, or if `'axis'` is None, a scalar is returned. If an output array is specified, a reference to `'out'` is returned.

#### See Also

-----

`ndarray.sum` : Equivalent method.

`cumsum` : Cumulative sum of array elements.

`trapz` : Integration of array values using the composite trapezoidal rule.

`mean`, `average`

#### Notes

-----

Arithmetic is modular when using `integer` types, and no error is raised on overflow.

The sum of an empty array is the neutral element 0:

```
>>> np.sum([])
0.0
```

#### Examples

```

>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])

If the accumulator is too small, overflow occurs:

>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128

swapaxes(a, axis1, axis2)
Interchange two axes of an array.

Parameters

a : array_like
 Input array.
axis1 : int
 First axis.
axis2 : int
 Second axis.

Returns

a_swapped : ndarray
 For Numpy >= 1.10, if `a` is an ndarray, then a view of `a` is
 returned; otherwise a new array is created. For earlier Numpy
 versions a view of `a` is returned only if the order of the
 axes is changed, otherwise the input array is returned.

Examples

>>> x = np.array([[1,2,3]])
>>> np.swapaxes(x,0,1)
array([[1,
 2,
 3]])

>>> x = np.array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])
>>> x
array([[[0, 1],
 [2, 3]],
 [[4, 5],
 [6, 7]]])

>>> np.swapaxes(x,0,2)
array([[[0, 4],
 [2, 6]],
 [[1, 5],
 [3, 7]]])

take(a, indices, axis=None, out=None, mode='raise')
Take elements from an array along an axis.

This function does the same thing as "fancy" indexing (indexing arrays
using arrays); however, it can be easier to use if you need elements
along a given axis.

Parameters

a : array_like
 The source array.
indices : array_like
 The indices of the values to extract.

.. versionadded:: 1.8.0

 Also allow scalars for indices.
axis : int, optional
 The axis over which to select values. By default, the flattened
 input array is used.
out : ndarray, optional
 If provided, the result will be placed in this array. It should
 be of the appropriate shape and dtype.
mode : {'raise', 'wrap', 'clip'}, optional
 Specifies how out-of-bounds indices will behave.

 * 'raise' -- raise an error (default)
 * 'wrap' -- wrap around
 * 'clip' -- clip to the range

```

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns

-----

`subarray : ndarray`  
The returned array has the same type as `a`.

See Also

-----

`compress` : Take elements using a boolean mask  
`ndarray.take` : equivalent method

Examples

-----

```
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> np.take(a, indices)
array([4, 3, 6])
```

In this example if `a` is an `ndarray`, "fancy" indexing can be used.

```
>>> a = np.array(a)
>>> a[indices]
array([4, 3, 6])
```

If `indices` is not one dimensional, the output also has these dimensions.

```
>>> np.take(a, [[0, 1], [2, 3]])
array([[4, 3],
 [5, 7]])
```

**tensordot(a, b, axes=2)**

Compute tensor dot product along specified axes for arrays >= 1-D.

Given two tensors (arrays of dimension greater than or equal to one), `a` and `b`, and an `array_like` `object` containing two `array_like` objects, ```(a_axes, b_axes)```, sum the products of `a`'s and `b`'s elements (components) over the axes specified by ```a_axes``` and ```b_axes```. The third argument can be a `single` non-negative integer\_like scalar, ```N``'; if it is such, then the last ```N``` dimensions of `a` and the first ```N``` dimensions of `b` are summed over.

Parameters

-----

`a, b : array_like, len(shape) >= 1`  
Tensors to "dot".

`axes : int or (2,) array_like`  
\* `integer_like`  
If an `int` `N`, sum over the last `N` axes of `a` and the first `N` axes of `b` in order. The sizes of the corresponding axes must match.  
\* `(2,) array_like`  
Or, a list of axes to be summed over, first sequence applying to `a`, second to `b`. Both elements `array_like` must be of the same length.

See Also

-----

`dot`, `einsum`

Notes

-----

Three common use cases are:

- ```axes = 0``` : tensor product `$a\otimes b$`
- ```axes = 1``` : tensor dot product `$a\cdot b$`
- ```axes = 2``` : (default) tensor `double` contraction `$a:b$`

When `axes` is `integer_like`, the sequence for evaluation will be: first the `-Nth` axis in `a` and `0th` axis in `b`, and the `-lth` axis in `a` and `Nth` axis in `b` last.

When there is more than one axis to sum over - and they are not the last (first) axes of `a` ('`b`') - the argument `axes` should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

Examples

-----

A "traditional" example:

```
>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> c = np.tensordot(a,b, axes=([1,0],[0,1]))
>>> c.shape
(5, 2)
>>> c
array([[4400., 4730.],
```

```

[4532., 4874.],
[4664., 5018.],
[4796., 5162.],
[4928., 5306.]])
>>> # A slower but equivalent way of computing the same...
>>> d = np.zeros((5,2))
>>> for i in range(5):
... for j in range(2):
... for k in range(3):
... for n in range(4):
... d[i,j] += a[k,n,i] * b[n,k,j]
>>> c == d
array([[True, True],
 [True, True],
 [True, True],
 [True, True],
 [True, True]], dtype=bool)

An extended example taking advantage of the overloading of + and *:

>>> a = np.array(range(1, 9))
>>> a.shape = (2, 2, 2)
>>> A = np.array('a', 'b', 'c', 'd'), dtype=object)
>>> A.shape = (2, 2)
>>> a; A
array([[[1, 2],
 [3, 4],
 [5, 6],
 [7, 8]]])
array([[a, b],
 [c, d]], dtype=object)

>>> np.tensordot(a, A) # third argument default is 2 for double-contraction
array([abbccddd, aaaaabbbbbbccccccddddd], dtype=object)

>>> np.tensordot(a, A, 1)
array([[[acc, bdd],
 [aaacccc, bbbdddd]],
 [[aaaaacccccc, bbbbbddddd],
 [aaaaaaaaaaaaaaaa, bbbbbbbddddd]]], dtype=object)

>>> np.tensordot(a, A, 0) # tensor product (result too long to incl.)
array([[[[a, b],
 [c, d]],
 ...
 ...
 ...
 [[[abbbb, cddddd],
 [aabbbb, ccddddd]],
 [[aaabbbbb, cccddddd],
 [aaaaabbbbb, ccccdffff]]], dtype=object)

>>> np.tensordot(a, A, (0, 1))
array([[[[ab, cdd],
 [aaabbb, cccddd]],
 [[aaabbbb, cccddd],
 [aaaaaaaaabbbbb, ccccccddddd]]], dtype=object)

>>> np.tensordot(a, A, (2, 1))
array([[[ab, cdd],
 [aaabbb, cccddd]],
 [[aaabbbb, cccddd],
 [aaaaaaaaabbbbb, ccccccddddd]]], dtype=object)

>>> np.tensordot(a, A, ((0, 1), (0, 1)))
array([abbbccccddddd, aabbcccccddddd], dtype=object)

>>> np.tensordot(a, A, ((2, 1), (1, 0)))
array([accbbddd, aaaaaccccccbbbbbddddd], dtype=object)

tile(A, reps)
Construct an array by repeating A the number of times given by reps.

If `reps` has length ``d``, the result will have dimension of
``max(d, A.ndim)``.

If ``A.ndim < d``, `A` is promoted to be d-dimensional by prepending new
axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication,
or shape (1, 1, 3) for 3-D replication. If this is not the desired
behavior, promote `A` to d-dimensions manually before calling this
function.

If ``A.ndim > d``, `reps` is promoted to `A`.ndim by pre-pending 1's to it.
Thus for an `A` of shape (2, 3, 4, 5), a `reps` of (2, 2) is treated as
(1, 1, 2, 2).

Note : Although tile may be used for broadcasting, it is strongly
recommended to use numpy's broadcasting operations and functions.

Parameters

A : array_like
 The input array.
reps : array_like
 The number of repetitions of `A` along each axis.

```

Returns  
-----  
c : [ndarray](#)  
    The tiled output array.

See Also  
-----  
[repeat](#) : Repeat elements of an array.  
[broadcast\\_to](#) : Broadcast an array to a new shape

Examples  
-----  

```
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
 [0, 1, 2, 0, 1, 2]])
>>> np.tile(a, (2, 1, 2))
array([[[0, 1, 2, 0, 1, 2]],
 [[0, 1, 2, 0, 1, 2]]])

>>> b = np.array([[1, 2], [3, 4]])
>>> np.tile(b, 2)
array([[1, 2, 1, 2],
 [3, 4, 3, 4]])
>>> np.tile(b, (2, 1))
array([[1, 2],
 [3, 4],
 [1, 2],
 [3, 4]])

>>> c = np.array([1, 2, 3, 4])
>>> np.tile(c, (4, 1))
array([[1, 2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 4]])
```

**trace**(a, offset=0, axis1=0, axis2=1, dtype=None, out=None)  
Return the sum along diagonals of the array.

If `a` is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements ``a[i,i+offset]`` for all i.

If `a` has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of `a` with `axis1` and `axis2` removed.

Parameters  
-----  
a : [array\\_like](#)  
    Input array, from which the diagonals are taken.  
offset : [int](#), optional  
    Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to 0.  
axis1, axis2 : [int](#), optional  
    Axes to be used as the first and second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults are the first two axes of `a`.  
dtype : [dtype](#), optional  
    Determines the data-type of the returned array and of the accumulator where the elements are summed. If [dtype](#) has the value None and `a` is of [integer](#) type of precision less than the default [integer](#) precision, then the default [integer](#) precision is used. Otherwise, the precision is the same as that of `a`.  
out : [ndarray](#), optional  
    Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output.

Returns  
-----  
sum\_along\_diagonals : [ndarray](#)  
    If `a` is 2-D, the sum along the diagonal is returned. If `a` has larger dimensions, then an array of sums along diagonals is returned.

See Also  
-----  
[diag](#), [diagonal](#), [diagflat](#)

Examples  
-----  

```
>>> np.trace(np.eye(3))
3.0
>>> a = np.arange(8).reshape((2,2,2))
>>> np.trace(a)
array([6, 8])
```

```
>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)

transpose(a, axes=None)
 Permute the dimensions of an array.

Parameters

a : array_like
 Input array.
axes : list of ints, optional
 By default, reverse the dimensions, otherwise permute the axes
 according to the values given.

Returns

p : ndarray
 'a' with its axes permuted. A view is returned whenever
 possible.

See Also

moveaxis
argsort

Notes

Use `transpose(a, argsort(axes))` to invert the transposition of tensors
when using the `axes` keyword argument.

Transposing a 1-D array returns an unchanged view of the original array.

Examples

>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
 [2, 3]])

>>> np.transpose(x)
array([[0, 2],
 [1, 3]])

>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)

trapz(y, x=None, dx=1.0, axis=-1)
 Integrate along the given axis using the composite trapezoidal rule.

 Integrate `y` ('x') along given axis.

Parameters

y : array_like
 Input array to integrate.
x : array_like, optional
 The sample points corresponding to the `y` values. If `x` is None,
 the sample points are assumed to be evenly spaced `dx` apart. The
 default is None.
dx : scalar, optional
 The spacing between sample points when `x` is None. The default is 1.
axis : int, optional
 The axis along which to integrate.

Returns

trapz : float
 Definite integral as approximated by trapezoidal rule.

See Also

sum, cumsum

Notes

Image [2]_ illustrates trapezoidal rule -- y-axis locations of points
will be taken from `y` array, by default x-axis distances between
points will be 1.0, alternatively they can be provided with `x` array
or with `dx` scalar. Return value will be equal to combined area under
the red lines.

References

.. [1] Wikipedia page: http://en.wikipedia.org/wiki/Trapezoidal_rule
```

```

.. [2] Illustration image:
 http://en.wikipedia.org/wiki/File:Composite_trapezoidal_rule_illustration.png

Examples

>>> np.trapz([1,2,3])
4.0
>>> np.trapz([1,2,3], x=[4,6,8])
8.0
>>> np.trapz([1,2,3], dx=2)
8.0
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
 [3, 4, 5]])
>>> np.trapz(a, axis=0)
array([1.5, 2.5, 3.5])
>>> np.trapz(a, axis=1)
array([2., 8.])

tri(N, M=None, k=0, dtype=<type 'float'>)
An array with ones at and below the given diagonal and zeros elsewhere.

Parameters

N : int
 Number of rows in the array.
M : int, optional
 Number of columns in the array.
 By default, 'M' is taken equal to 'N'.
k : int, optional
 The sub-diagonal at and below which the array is filled.
 'k' = 0 is the main diagonal, while 'k' < 0 is below it,
 and 'k' > 0 is above. The default is 0.
dtype : dtype, optional
 Data type of the returned array. The default is float.

Returns

tri : ndarray of shape (N, M)
 Array with its lower triangle filled with ones and zero elsewhere;
 in other words ``T[i,j] == 1`` for ``i <= j + k``, 0 otherwise.

Examples

>>> np.tri(3, 5, 2, dtype=int)
array([[1, 1, 1, 0, 0],
 [1, 1, 1, 1, 0],
 [1, 1, 1, 1, 1]])

>>> np.tri(3, 5, -1)
array([[0., 0., 0., 0., 0.],
 [1., 0., 0., 0., 0.],
 [1., 1., 0., 0., 0.]))

tril(m, k=0)
Lower triangle of an array.

Return a copy of an array with elements above the `k`-th diagonal zeroed.

Parameters

m : array_like, shape (M, N)
 Input array.
k : int, optional
 Diagonal above which to zero elements. `k = 0` (the default) is the
 main diagonal, `k < 0` is below it and `k > 0` is above.

Returns

tril : ndarray, shape (M, N)
 Lower triangle of `m`, of same shape and data-type as `m`.

See Also

triu : same thing, only for the upper triangle

Examples

>>> np.tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[0, 0, 0],
 [4, 0, 0],
 [7, 8, 0],
 [10, 11, 12]])

tril_indices(n, k=0, m=None)
Return the indices for the lower-triangle of an (n, m) array.

```

```

Parameters

n : int
 The row dimension of the arrays for which the returned
 indices will be valid.
k : int, optional
 Diagonal offset (see `tril` for details).
m : int, optional
 .. versionadded:: 1.9.0

 The column dimension of the arrays for which the returned
 arrays will be valid.
 By default `m` is taken equal to `n`.

Returns

inds : tuple of arrays
 The indices for the triangle. The returned tuple contains two arrays,
 each with the indices along one dimension of the array.

See also

triu_indices : similar function, for upper-triangular.
mask_indices : generic function accepting an arbitrary mask function.
tril, triu

Notes

.. versionadded:: 1.4.0

Examples

Compute two different sets of indices to access 4x4 arrays, one for the
lower triangular part starting at the main diagonal, and one starting two
diagonals further right:

>>> il1 = np.tril_indices(4)
>>> il2 = np.tril_indices(4, 2)

Here is how they can be used with a sample array:

>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11],
 [12, 13, 14, 15]])

Both for indexing:

>>> a[il1]
array([0, 4, 5, 8, 9, 10, 12, 13, 14, 15])

And for assigning values:

>>> a[il1] = -1
>>> a
array([[-1, 1, 2, 3],
 [-1, -1, 6, 7],
 [-1, -1, -1, 11],
 [-1, -1, -1, -1]])

These cover almost the whole array (two diagonals right of the main one):

>>> a[il2] = -10
>>> a
array([[-10, -10, -10, 3],
 [-10, -10, -10, -10],
 [-10, -10, -10, -10],
 [-10, -10, -10, -10]])

tril_indices_from(arr, k=0)
Return the indices for the lower-triangle of arr.

See `tril_indices` for full details.

Parameters

arr : array_like
 The indices will be valid for square arrays whose dimensions are
 the same as arr.
k : int, optional
 Diagonal offset (see `tril` for details).

See Also

tril_indices, tril

```

**Notes**  
 -----  
 .. versionadded:: 1.4.0

**trim\_zeros(filt, trim='fb')**  
 Trim the leading and/or trailing zeros from a 1-D array or sequence.

**Parameters**  
 -----  
 filt : 1-D array or sequence  
 Input array.  
 trim : [str](#), optional  
 A string with 'f' representing trim from front and 'b' to trim from back. Default is 'fb', trim zeros from both front and back of the array.

**Returns**  
 -----  
 trimmed : 1-D array or sequence  
 The result of trimming the input. The input data type is preserved.

**Examples**  
 -----  
`>>> a = np.array([0, 0, 0, 1, 2, 3, 0, 2, 1, 0])
>>> np.trim_zeros(a)
array([1, 2, 3, 0, 2, 1])`  
`>>> np.trim_zeros(a, 'b')
array([0, 0, 0, 1, 2, 3, 0, 2, 1])`

The input data type is preserved, list/tuple in means list/tuple out.

`>>> np.trim_zeros([0, 1, 2, 0])
[1, 2]`

**triu(m, k=0)**  
 Upper triangle of an array.

Return a copy of a [matrix](#) with the elements below the `k`-th diagonal zeroed.

Please refer to the documentation for `tril` for further details.

**See Also**  
 -----  
 tril : lower triangle of an array

**Examples**  
 -----  
`>>> np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 1, 2, 3],
 [ 4, 5, 6],
 [ 0, 8, 9],
 [ 0, 0, 12]])`

**triu\_indices(n, k=0, m=None)**  
 Return the indices for the upper-triangle of an (n, m) array.

**Parameters**  
 -----  
 n : [int](#)  
 The size of the arrays for which the returned indices will be valid.  
 k : [int](#), optional  
 Diagonal offset (see `triu` for details).  
 m : [int](#), optional  
 .. versionadded:: 1.9.0

The column dimension of the arrays for which the returned arrays will be valid.  
 By default `m` is taken equal to `n`.

**Returns**  
 -----  
 inds : tuple, [shape](#)(2) of ndarrays, [shape](#)('n')  
 The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array. Can be used to slice a [ndarray](#) of [shape](#)('n', 'n').

**See also**  
 -----  
 tril\_indices : similar function, for lower-triangular.  
 mask\_indices : [generic](#) function accepting an arbitrary mask function.  
 triu, tril

**Notes**

```

.. versionadded:: 1.4.0

Examples

Compute two different sets of indices to access 4x4 arrays, one for the
upper triangular part starting at the main diagonal, and one starting two
diagonals further right:
```

```
>>> iu1 = np.triu_indices(4)
>>> iu2 = np.triu_indices(4, 2)
```

Here is how they can be used with a sample array:

```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11],
 [12, 13, 14, 15]])
```

Both for indexing:

```
>>> a[iu1]
array([0, 1, 2, 3, 5, 6, 7, 10, 11, 15])
```

And for assigning values:

```
>>> a[iu1] = -1
>>> a
array([[-1, -1, -1, -1],
 [4, -1, -1, -1],
 [8, 9, -1, -1],
 [12, 13, 14, -1]])
```

These cover only a small part of the whole array (two diagonals right
of the main one):

```
>>> a[iu2] = -10
>>> a
array([[-1, -1, -10, -10],
 [4, -1, -1, -10],
 [8, 9, -1, -1],
 [12, 13, 14, -1]])
```

## triu\_indices\_from(arr, k=0)

Return the indices for the upper-triangle of arr.

See `triu\_indices` for full details.

Parameters

```

arr : ndarray, shape(N, N)
 The indices will be valid for square arrays.
k : int, optional
 Diagonal offset (see `triu` for details).
```

Returns

```

triu_indices_from : tuple, shape(2) of ndarray, shape(N)
 Indices for the upper-triangle of `arr`.
```

See Also

-----
triu\_indices, triu

Notes

```

.. versionadded:: 1.4.0
```

## typename(char)

Return a description for the given data type code.

Parameters

```

char : str
 Data type code.
```

Returns

```

out : str
 Description of the input data type code.
```

See Also

-----
dtype, typecodes

Examples

```

>>> typechars = ['S1', '?', 'B', 'D', 'G', 'F', 'I', 'H', 'L', 'O', 'Q',
... 'S', 'U', 'V', 'b', 'd', 'g', 'f', 'i', 'h', 'l', 'q']
>>> for typechar in typechars:
... print(typechar, ' : ', np.typename(typechar))
...
S1 : character
? : bool
B : unsigned char
D : complex double precision
G : complex long double precision
F : complex single precision
I : unsigned integer
H : unsigned short
L : unsigned long integer
O : object
Q : unsigned long long integer
S : string
U : unicode
V : void
b : signed char
d : double precision
g : long precision
f : single precision
i : integer
h : short
l : long integer
q : long long integer

union1d(ar1, ar2)
Find the union of two arrays.

Return the unique, sorted array of values that are in either of the two
input arrays.

Parameters

ar1, ar2 : array_like
 Input arrays. They are flattened if they are not already 1D.

Returns

union1d : ndarray
 Unique, sorted union of the input arrays.

See Also

numpy.lib.arraysetops : Module with a number of other functions for
 performing set operations on arrays.

Examples

>>> np.union1d([-1, 0, 1], [-2, 0, 2])
array([-2, -1, 0, 1, 2])

To find the union of more than two arrays, use functools.reduce:

>>> from functools import reduce
>>> reduce(np.union1d, ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2]))
array([1, 2, 3, 4, 6])

unique(ar, return_index=False, return_inverse=False, return_counts=False)
Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional
outputs in addition to the unique elements: the indices of the input array
that give the unique values, the indices of the unique array that
reconstruct the input array, and the number of times each unique value
comes up in the input array.

Parameters

ar : array_like
 Input array. This will be flattened if it is not already 1-D.
return_index : bool, optional
 If True, also return the indices of `ar` that result in the unique
 array.
return_inverse : bool, optional
 If True, also return the indices of the unique array that can be used
 to reconstruct `ar`.
return_counts : bool, optional
 If True, also return the number of times each unique value comes up
 in `ar`.

.. versionadded:: 1.9.0

Returns

```

`unique : ndarray`  
     The sorted unique values.  
`unique_indices : ndarray, optional`  
     The indices of the first occurrences of the unique values in the  
     (flattened) original array. Only provided if `return\_index` is True.  
`unique_inverse : ndarray, optional`  
     The indices to reconstruct the (flattened) original array from the  
     unique array. Only provided if `return\_inverse` is True.  
`unique_counts : ndarray, optional`  
     The `number` of times each of the unique values comes up in the  
     original array. Only provided if `return\_counts` is True.

.. versionadded:: 1.9.0

#### See Also

-----  
`numpy.lib.arraysetops` : Module with a `number` of other functions for  
     performing set operations on arrays.

#### Examples

-----  
`>>> np.unique([1, 1, 2, 2, 3, 3])`  
`array([1, 2, 3])`  
`>>> a = np.array([[1, 1], [2, 3]])`  
`>>> np.unique(a)`  
`array([1, 2, 3])`

Return the indices of the original array that give the unique values:

-----  
`>>> a = np.array(['a', 'b', 'b', 'c', 'a'])`  
`>>> u, indices = np.unique(a, return_index=True)`  
`>>> u`  
`array(['a', 'b', 'c'],`  
`dtype='|S1')`  
`>>> indices`  
`array([0, 1, 3])`  
`>>> a[indices]`  
`array(['a', 'b', 'c'],`  
`dtype='|S1')`

Reconstruct the input array from the unique values:

-----  
`>>> a = np.array([1, 2, 6, 4, 2, 3, 2])`  
`>>> u, indices = np.unique(a, return_inverse=True)`  
`>>> u`  
`array([1, 2, 3, 4, 6])`  
`>>> indices`  
`array([0, 1, 4, 3, 1, 2, 1])`  
`>>> u[indices]`  
`array([1, 2, 6, 4, 2, 3, 2])`

### unpackbits(...)

`unpackbits(myarray, axis=None)`

Unpacks elements of a `uint8` array into a binary-valued output array.

Each element of `myarray` represents a bit-field that should be unpacked  
     into a binary-valued output array. The shape of the output array is either  
     1-D (if `axis` is None) or the same shape as the input array with unpacking  
     done along the axis specified.

#### Parameters

-----  
`myarray : ndarray, uint8 type`  
     Input array.  
`axis : int, optional`  
     Unpacks along this axis.

#### Returns

-----  
`unpacked : ndarray, uint8 type`  
     The elements are binary-valued (0 or 1).

#### See Also

-----  
`packbits` : Packs the elements of a binary-valued array into bits in a `uint8`  
     array.

#### Examples

-----  
`>>> a = np.array([[2], [7], [23]], dtype=np.uint8)`  
`>>> a`  
`array([[ 2],`  
`[ 7],`  
`[23]], dtype=uint8)`  
`>>> b = np.unpackbits(a, axis=1)`  
`>>> b`  
`array([[0, 0, 0, 0, 0, 0, 1, 0],`  
`[0, 0, 0, 0, 1, 1, 1, 0],`

```
[0, 0, 0, 1, 0, 1, 1, 1]], dtype=uint8)
```

**unravel\_index(...)**

```
unravel_index(indices, dims, order='C')
```

Converts a flat index or array of flat indices into a tuple of coordinate arrays.

Parameters

-----

- indices : array\_like
 An [integer](#) array whose elements are indices into the flattened version of an array of dimensions ``dims''. Before version 1.6.0, this function accepted just one index value.
- dims : tuple of ints
 The shape of the array to use for unraveling ``indices''.
- order : {'C', 'F'}, optional
 Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

.. versionadded:: 1.6.0

Returns

-----

- unraveled\_coords : tuple of [ndarray](#)
 Each array in the tuple has the same shape as the ``indices'' array.

See Also

-----

- [ravel\\_multi\\_index](#)

Examples

-----

```
>>> np.unravel_index([22, 41, 37], (7,6))
(array([3, 6, 6]), array([4, 5, 1]))
>>> np.unravel_index([31, 41, 13], (7,6), order='F')
(array([3, 6, 6]), array([4, 5, 1]))

>>> np.unravel_index(1621, (6,7,8,9))
(3, 1, 4, 1)
```

**unwrap(p, discont=3.141592653589793, axis=-1)**

Unwrap by changing deltas between values to 2\*pi complement.

Unwrap radian phase `p` by changing absolute jumps greater than `discont` to their 2\*pi complement along the given axis.

Parameters

-----

- p : array\_like
 Input array.
- discont : [float](#), optional
 Maximum discontinuity between values, default is ``pi``.
- axis : [int](#), optional
 Axis along which unwrap will operate, default is the last axis.

Returns

-----

- out : [ndarray](#)
 Output array.

See Also

-----

- [rad2deg](#), [deg2rad](#)

Notes

-----

If the discontinuity in `p` is smaller than ``pi``, but larger than `discont`, no unwrapping is done because taking the 2\*pi complement would only make the discontinuity larger.

Examples

-----

```
>>> phase = np.linspace(0, np.pi, num=5)
>>> phase[3:] += np.pi
>>> phase
array([0. , 0.78539816, 1.57079633, 5.49778714, 6.28318531])
>>> np.unwrap(phase)
array([0. , 0.78539816, 1.57079633, -0.78539816, 0.])
```

**vander(x, N=None, increasing=False)**

Generate a Vandermonde [matrix](#).

The columns of the output [matrix](#) are powers of the input vector. The order of the powers is determined by the `increasing` boolean argument. Specifically, when `increasing` is False, the `i`-th output column is the input vector raised element-wise to the power of ``N - i - 1``. Such

a [matrix](#) with a geometric progression in each row is named for Alexandre-Theophile Vandermonde.

**Parameters**

-----

`x` : [array\\_like](#)

    1-D input array.

`N` : [int](#), optional

    Number of columns in the output. If `N` is not specified, a square array is returned (``N = len(x)`').

`increasing` : [bool](#), optional

    Order of the powers of the columns. If True, the powers increase from left to right, if False (the default) they are reversed.

.. versionadded:: 1.9.0

**Returns**

-----

`out` : [ndarray](#)

    Vandermonde [matrix](#). If `increasing` is False, the first column is `` $x^{(N-1)}$ `` , the second `` $x^{(N-2)}$ `` and so forth. If `increasing` is True, the columns are `` $x^0, x^1, \dots, x^{(N-1)}$ ``.

**See Also**

-----

[polynomial.polynomial.polyvander](#)

**Examples**

-----

>>> x = np.array([1, 2, 3, 5])

>>> N = 3

>>> np.vander(x, N)

[array](#)([[ 1, 1, 1],  
[ 4, 2, 1],  
[ 9, 3, 1],  
[25, 5, 1]])

>>> np.column\_stack([x\*\*(N-1-i) for i in range(N)])

[array](#)([[ 1, 1, 1],  
[ 4, 2, 1],  
[ 9, 3, 1],  
[25, 5, 1]])

>>> x = np.array([1, 2, 3, 5])

>>> np.vander(x)

[array](#)([[ 1, 1, 1, 1],  
[ 8, 4, 2, 1],  
[ 27, 9, 3, 1],  
[125, 25, 5, 1]])

>>> np.vander(x, increasing=True)

[array](#)([[ 1, 1, 1, 1],  
[ 1, 2, 4, 8],  
[ 1, 3, 9, 27],  
[ 1, 5, 25, 125]])

The determinant of a square Vandermonde [matrix](#) is the product of the differences between the values of the input vector:

>>> np.linalg.det(np.vander(x))

48.00000000000043

>>> (5-3)\*(5-2)\*(5-1)\*(3-2)\*(3-1)\*(2-1)

48

**var**(`a`, `axis=None`, `dtype=None`, `out=None`, `ddof=0`, `keepdims=False`)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

-----

`a` : [array\\_like](#)

    Array containing numbers whose variance is desired. If `a` is not an array, a conversion is attempted.

`axis` : `None` or [int](#) or tuple of ints, optional

    Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

.. versionadded: 1.7.0

If this is a tuple of ints, a variance is performed over multiple axes, instead of a [single](#) axis or all the axes as before.

`dtype` : data-type, optional

    Type to use in computing the variance. For arrays of [integer](#) type the default is `'float32'`; for arrays of [float](#) types it is the same as the array type.

`out` : [ndarray](#), optional

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

the same shape as the expected output, but the type is cast if necessary.

`ddof` : `int`, optional  
 "Delta Degrees of Freedom": the divisor used in the calculation is `` $N - ddof$ '', where `` $N$ '' represents the `number` of elements. By default `ddof` is zero.

`keepdims` : `bool`, optional  
 If this is set to True, the axes which are reduced are left in the result as dimensions of size one. With this option, the result will `broadcast` correctly against the original `arr`.

Returns

-----

`variance` : `ndarray`, see `dtype` parameter above  
 If ``out=None'', returns a new array containing the variance; otherwise, a reference to the output array is returned.

See Also

-----

`std` , `mean`, `nanmean`, `nanstd`, `nanvar`  
`numpy.doc.ufuncs` : Section "Output arguments"

Notes

-----

The variance is the average of the squared deviations from the mean, i.e., `` $\text{var} = \text{mean}(\text{abs}(x - x.\text{mean})^2)$ ''.

The mean is normally calculated as `` $x.\text{sum}() / N$ '', where `` $N = \text{len}(x)$ ''. If, however, `ddof` is specified, the divisor `` $N - ddof$ '' is used instead. In standard statistical practice, ```ddof=1`'' provides an unbiased estimator of the variance of a hypothetical infinite population. ```ddof=0`'' provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for `complex` numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For `floating`-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-accuracy accumulator using the ```dtype``` keyword can alleviate this issue.

Examples

-----

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1., 1.])
>>> np.var(a, axis=1)
array([0.25, 0.25])
```

In `single` precision, `var()` can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.var(a)
0.20250003
```

Computing the variance in `float64` is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932944759
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025
```

**vdot(...)**

`vdot`(`a`, `b`)

Return the dot product of two vectors.

The `vdot('a', 'b')` function handles `complex` numbers differently than `dot('a', 'b')`. If the first argument is `complex` the `complex` conjugate of the first argument is used for the calculation of the dot product.

Note that `vdot` handles multidimensional arrays differently than `dot`: it does \*not\* perform a `matrix` product, but flattens input arguments to 1-D vectors first. Consequently, it should only be used for vectors.

Parameters

-----

`a` : `array_like`  
 If `a` is `complex` the `complex` conjugate is taken before calculation of the dot product.

`b` : `array_like`  
 Second argument to the dot product.

**Returns**  
-----  
**output : ndarray**  
Dot product of `a` and `b`. Can be an [int](#), [float](#), or [complex](#) depending on the types of `a` and `b`.

**See Also**  
-----  
**dot** : Return the dot product without using the [complex](#) conjugate of the first argument.

**Examples**  
-----  
`>>> a = np.array([1+2j, 3+4j])  
>>> b = np.array([5+6j, 7+8j])  
>>> np.vdot(a, b)  
(70-8j)  
>>> np.vdot(b, a)  
(70+8j)`

Note that higher-dimensional arrays are flattened!

`>>> a = np.array([[1, 4], [5, 6]])  
>>> b = np.array([[4, 1], [2, 2]])  
>>> np.vdot(a, b)  
30  
>>> np.vdot(b, a)  
30  
>>> 1*4 + 4*1 + 5*2 + 6*2  
30`

**vsplit(ary, indices\_or\_sections)**  
Split an array into multiple sub-arrays vertically (row-wise).

Please refer to the ``split`` documentation. ``vsplit`` is equivalent to ``split`` with `axis=0` (default), the array is always split along the first axis regardless of the array dimension.

**See Also**  
-----  
**split** : Split an array into multiple sub-arrays of equal size.

**Examples**  
-----  
`>>> x = np.arange(16.0).reshape(4, 4)  
>>> x  
array([[ 0., 1., 2., 3.],  
 [ 4., 5., 6., 7.],  
 [ 8., 9., 10., 11.],  
 [12., 13., 14., 15.]])  
>>> np.vsplit(x, 2)  
[array([[ 0., 1., 2., 3.],  
 [ 4., 5., 6., 7.]]),  
 array([[ 8., 9., 10., 11.],  
 [12., 13., 14., 15.]])]  
>>> np.vsplit(x, np.array([3, 6]))  
[array([[ 0., 1., 2., 3.],  
 [ 4., 5., 6., 7.],  
 [ 8., 9., 10., 11.]]),  
 array([[12., 13., 14., 15.]]),  
 array([], dtype=float64)]`

With a higher dimensional array the split is still along the first axis.

`>>> x = np.arange(8.0).reshape(2, 2, 2)  
>>> x  
array([[[ 0., 1.],  
 [ 2., 3.]],  
 [[ 4., 5.],  
 [ 6., 7.]]])  
>>> np.vsplit(x, 2)  
[array([[[ 0., 1.],  
 [ 2., 3.]]]),  
 array([[[ 4., 5.],  
 [ 6., 7.]]])]`

**vstack(tup)**  
Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a [single](#) array. Rebuild arrays divided by `vsplit`.

**Parameters**  
-----  
**tup : sequence of ndarrays**  
Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns  
-----  
`stacked : ndarray`  
The array formed by stacking the given arrays.

## See Also

`stack` : Join a sequence of arrays along a new axis.  
`hstack` : Stack arrays in sequence horizontally (column wise).  
`dstack` : Stack arrays in sequence depth wise (along third dimension).  
`concatenate` : Join a sequence of arrays along an existing axis.  
`vsplit` : Split array into a list of multiple sub-arrays vertically.

## Notes

-----  
Equivalent to ```np.concatenate(tup, axis=0)``` if `tup` contains arrays that are at least 2-dimensional.

## Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
 [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
 [2],
 [3],
 [2],
 [3],
 [4]])
```

**where(...)**

`where(condition, [x, y])`

Return elements, either from `x` or `y`, depending on `condition`.

If only `condition` is given, return ```condition.nonzero()```.

## Parameters

-----  
`condition : array_like, bool`  
When True, yield `x`, otherwise yield `y`.  
`x, y : array_like, optional`  
Values from which to choose. `x` and `y` need to have the same shape as `condition`.

## Returns

-----  
`out : ndarray or tuple of ndarrays`  
If both `x` and `y` are specified, the output array contains elements of `x` where `condition` is True, and elements from `y` elsewhere.  
If only `condition` is given, return the tuple ```condition.nonzero()``` , the indices where `condition` is True.

## See Also

-----  
`nonzero, choose`

## Notes

-----  
If `x` and `y` are given and input arrays are 1-D, `where` is equivalent to::

```
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

## Examples

```
>>> np.where([[True, False], [True, True]],
... [[1, 2], [3, 4]],
... [[9, 8], [7, 6]])
array([[1, 8],
 [3, 4]])

>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))

>>> x = np.arange(9.).reshape(3, 3)
>>> np.where(x > 5)
(array([2, 2, 2]), array([0, 1, 2])) # Note: result is 1D.
>>> x[np.where(x > 3.0)]
array([4., 5., 6., 7., 8.]) # Note: broadcasting.
>>> np.where(x < 5, x, -1)
```

```

array([[0., 1., 2.],
 [3., 4., -1.],
 [-1., -1., -1.]])
```

Find the indices of elements of `x` that are in `goodvalues`.

```

>>> goodvalues = [3, 4, 7]
>>> ix = np.in1d(x.ravel(), goodvalues).reshape(x.shape)
>>> ix
array([[False, False, False],
 [True, True, False],
 [False, True, False]], dtype=bool)
>>> np.where(ix)
(array([1, 1, 2]), array([0, 1, 1]))
```

**who(vardict=None)**

Print the Numpy arrays in the given dictionary.

If there is no dictionary passed in or `vardict` is None then returns Numpy arrays in the `globals()` dictionary (all Numpy arrays in the namespace).

**Parameters**

-----

`vardict : dict, optional`  
A dictionary possibly containing ndarrays. Default is `globals()`.

**Returns**

-----

`out : None`  
Returns 'None'.

**Notes**

-----

Prints out the name, shape, bytes and type of all of the ndarrays present in `vardict`.

**Examples**

-----

```

>>> a = np.arange(10)
>>> b = np.ones(20)
>>> np.who()
Name Shape Bytes Type
=====
a 10 40 int32
b 20 160 float64
Upper bound on total bytes = 200
```

```

>>> d = {'x': np.arange(2.0), 'y': np.arange(3.0), 'txt': 'Some str',
... 'idx':5}
>>> np.who(d)
Name Shape Bytes Type
=====
y 3 24 float64
x 2 16 float64
Upper bound on total bytes = 40
```

**zeros(...)**

`zeros(shape, dtype=float, order='C')`

Return a new array of given shape and type, filled with zeros.

**Parameters**

-----

`shape : int or sequence of ints`  
Shape of the new array, e.g., `(2, 3)` or `2`.

`dtype : data-type, optional`  
The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

`order : {'C', 'F'}, optional`  
Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

-----

`out : ndarray`  
Array of zeros with the given shape, `dtype`, and order.

**See Also**

-----

`zeros_like` : Return an array of zeros with shape and type of input.  
`ones_like` : Return an array of ones with shape and type of input.  
`empty_like` : Return an empty array with shape and type of input.  
`ones` : Return a new array setting values to one.  
`empty` : Return a new uninitialized array.

**Examples**

-----

```

>>> np.zeros(5)
array([0., 0., 0., 0., 0.])

>>> np.zeros((5,), dtype=np.int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[0.],
 [0.]])
```

>>> s = (2,2)

```
>>> np.zeros(s)
array([[0., 0.],
 [0., 0.]])
```

>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
 dtype=[('x', '<i4'), ('y', '<i4')])

**zeros\_like**(a, dtype=None, order='K', subok=True)

Return an array of zeros with the same shape and type as a given array.

Parameters

a : array\_like  
The shape and data-type of `a` define these same attributes of the returned array.

dtype : data-type, optional  
Overrides the data type of the result.

.. versionadded:: 1.6.0  
order : {'C', 'F', 'A', or 'K'}, optional  
Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.

.. versionadded:: 1.6.0  
subok : bool, optional.  
If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

Returns

out : ndarray  
Array of zeros with the same shape and type as `a`.

See Also

ones\_like : Return an array of ones with shape and type of input.  
empty\_like : Return an empty array with shape and type of input.  
zeros : Return a new array setting values to zero.  
ones : Return a new array setting values to one.  
empty : Return a new uninitialized array.

Examples

```

>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
 [3, 4, 5]])
>>> np.zeros_like(x)
array([[0, 0, 0],
 [0, 0, 0]])

>>> y = np.arange(3, dtype=np.float)
>>> y
array([0., 1., 2.])
>>> np.zeros_like(y)
array([0., 0., 0.])
```

## Data

```

ALLOW_THREADS = 1
BUFSIZE = 8192
CLIP = 0
ERR_CALL = 3
ERR_DEFAULT = 521
ERR_IGNORE = 0
ERR_LOG = 5
ERR_PRINT = 4
ERR_RAISE = 2
```

```
ERR_WARN = 1
FLOATING_POINT_SUPPORT = 1
FPE_DIVIDEBYZERO = 1
FPE_INVALID = 8
FPE_OVERFLOW = 2
FPE_UNDERFLOW = 4
False_ = False
Inf = inf
Infinity = inf
MAXDIMS = 32
MAY_SHARE_BOUNDS = 0
MAY_SHARE_EXACT = -1
NAN = nan
NINF = -inf
NZERO = -0.0
NaN = nan
PINF = inf
PZERO = 0.0
RAISE = 2
SHIFT_DIVIDEBYZERO = 0
SHIFT_INVALID = 9
SHIFT_OVERFLOW = 3
SHIFT_UNDERFLOW = 6
ScalarType = (<type 'int'>, <type 'float'>, <type 'complex'>, <type 'long'>, <type 'bool'>, <type 'str'>, <type 'unicode'>, <type 'buffer'>, <type 'numpy.bool_'>, <type 'numpy.timedelta64'>, <type 'numpy.float16'>, <type 'numpy.uint8'>, <type 'numpy.int8'>, <type 'numpy.object_'>, <type 'numpy.datetime64'>, <type 'numpy.uint64'>, <type 'numpy.int64'>, <type 'numpy.void'>, <type 'numpy.complex256'>, <type 'numpy.float128'>, ...)
True_ = True
UFUNC_BUFSIZE_DEFAULT = 8192
UFUNC_PYVALS_NAME = 'UFUNC_PYVALS'
WRAP = 1
_NUMPY_SETUP_ = False
all = ['add_newdocs', 'ModuleDeprecationWarning', 'VisibleDeprecationWarning', '__version__', 'pkgload', 'PackageLoader', 'show_config', 'char', 'rec', 'memmap', 'newaxis', 'ndarray', 'flatiter', 'nditer', 'nested_iters', 'ufunc', 'arange', 'array', 'zeros', 'count_nonzero', ...]
_git_revision_ = '4092a9e160cc247a4a45724579a0c829733688ca'
version = '1.11.0'
absolute = <ufunc 'absolute'>
add = <ufunc 'add'>
arccos = <ufunc 'arccos'>
arccosh = <ufunc 'arccosh'>
arcsin = <ufunc 'arcsin'>
arcsinh = <ufunc 'arcsinh'>
arctan = <ufunc 'arctan'>
arctan2 = <ufunc 'arctan2'>
arctanh = <ufunc 'arctanh'>
bitwise_and = <ufunc 'bitwise_and'>
bitwise_not = <ufunc 'invert'>
bitwise_or = <ufunc 'bitwise_or'>
bitwise_xor = <ufunc 'bitwise_xor'>
c_ = <numpy.lib.index_tricks.CClass object>
cast = {<type 'numpy.bool_'>: <function <lambda> at 0x7...y.int16'>: <function <lambda> at 0x7fea52b2f398>}
cbrt = <ufunc 'cbrt'>
ceil = <ufunc 'ceil'>
conj = <ufunc 'conjugate'>
conjugate = <ufunc 'conjugate'>
copysign = <ufunc 'copysign'>
cos = <ufunc 'cos'>
cosh = <ufunc 'cosh'>
deg2rad = <ufunc 'deg2rad'>
degrees = <ufunc 'degrees'>
divide = <ufunc 'divide'>
e = 2.718281828459045
equal = <ufunc 'equal'>
euler_gamma = 0.5772156649015329
exp = <ufunc 'exp'>
exp2 = <ufunc 'exp2'>
expm1 = <ufunc 'expm1'>
```

```
fabs = <ufunc 'fabs'>
floor = <ufunc 'floor'>
floor_divide = <ufunc 'floor_divide'>
fmax = <ufunc 'fmax'>
fmin = <ufunc 'fmin'>
fmod = <ufunc 'fmod'>
frexp = <ufunc 'frexp'>
greater = <ufunc 'greater'>
greater_equal = <ufunc 'greater_equal'>
hypot = <ufunc 'hypot'>
index_exp = <numpy.lib.index_tricks.IndexExpression object>
inf = inf
infty = inf
invert = <ufunc 'invert'>
isfinite = <ufunc 'isfinite'>
isinf = <ufunc 'isinf'>
isnan = <ufunc 'isnan'>
ldexp = <ufunc 'ldexp'>
left_shift = <ufunc 'left_shift'>
less = <ufunc 'less'>
less_equal = <ufunc 'less_equal'>
little_endian = True
log = <ufunc 'log'>
log10 = <ufunc 'log10'>
log1p = <ufunc 'log1p'>
log2 = <ufunc 'log2'>
logaddexp = <ufunc 'logaddexp'>
logaddexp2 = <ufunc 'logaddexp2'>
logical_and = <ufunc 'logical_and'>
logical_not = <ufunc 'logical_not'>
logical_or = <ufunc 'logical_or'>
logical_xor = <ufunc 'logical_xor'>
maximum = <ufunc 'maximum'>
mgrid = <numpy.lib.index_tricks.nd_grid object>
minimum = <ufunc 'minimum'>
mod = <ufunc 'remainder'>
modf = <ufunc 'modf'>
multiply = <ufunc 'multiply'>
nan = nan
nbytes = {<type 'numpy.bool_': 1, <type 'numpy.timedelta64': 2, <type 'numpy.uint16': 2, <type 'numpy.int16': 2}
negative = <ufunc 'negative'>
newaxis = None
nextafter = <ufunc 'nextafter'>
not_equal = <ufunc 'not_equal'>
ogrid = <numpy.lib.index_tricks.nd_grid object>
pi = 3.141592653589793
power = <ufunc 'power'>
r_ = <numpy.lib.index_tricks.RClass object>
rad2deg = <ufunc 'rad2deg'>
radians = <ufunc 'radians'>
reciprocal = <ufunc 'reciprocal'>
remainder = <ufunc 'remainder'>
right_shift = <ufunc 'right_shift'>
rint = <ufunc 'rint'>
s_ = <numpy.lib.index_tricks.IndexExpression object>
sctypeDict = {0: <type 'numpy.bool_>, 1: <type 'numpy.int8'>, 2: <type 'numpy.uint8'>, 3: <type 'numpy.int16'>, 4: <type 'numpy.uint16'>, 5: <type 'numpy.int32'>, 6: <type 'numpy.uint32'>, 7: <type 'numpy.int64'>, 8: <type 'numpy.uint64'>, 9: <type 'numpy.int64'>, ...}
sctypeNA = {'?': 'Bool', 'B': 'UInt8', 'Bool': <type 'numpy.bool_>, 'Complex128': <type 'numpy.complex256'>, 'Complex32': <type 'numpy.complex64'>, 'Complex64': <type 'numpy.complex128'>, 'D': 'Complex64', 'Datetime64': <type 'numpy.datetime64'>, 'F': 'Complex32', 'Float128': <type 'numpy.float128'>, ...}
sctypes = {'complex': [<type 'numpy.complex64'>, <type 'numpy.complex128'>, <type 'numpy.complex256'>], 'float': [<type 'numpy.float16'>, <type 'numpy.float32'>, <type 'numpy.float64'>, <type 'numpy.float128'>], 'int': [<type 'numpy.int8'>, <type 'numpy.int16'>, <type 'numpy.int32'>, <type 'numpy.int64'>], 'others': [<type 'bool'>, <type 'object'>, <type 'str'>, <type 'unicode'>, <type 'numpy.void'>], 'uint': [<type 'numpy.uint8'>, <type 'numpy.uint16'>, <type 'numpy.uint32'>, <type 'numpy.uint64'>]}}
sign = <ufunc 'sign'>
signbit = <ufunc 'signbit'>
sin = <ufunc 'sin'>
sinh = <ufunc 'sinh'>
```

```
spacing = <ufunc 'spacing'>
sqrt = <ufunc 'sqrt'>
square = <ufunc 'square'>
subtract = <ufunc 'subtract'>
tan = <ufunc 'tan'>
tanh = <ufunc 'tanh'>
true_divide = <ufunc 'true_divide'>
trunc = <ufunc 'trunc'>
typeDict = {0: <type 'numpy.bool_'>, 1: <type 'numpy.int8'>, 2: <type 'numpy.uint8'>, 3: <type 'numpy.int16'>, 4: <type 'numpy.uint16'>, 5: <type 'numpy.int32'>, 6: <type 'numpy.uint32'>, 7: <type 'numpy.int64'>, 8: <type 'numpy.uint64'>, 9: <type 'numpy.int64'>, ...}
typeNA = {'?': 'Bool', 'B': 'UInt8', 'Bool': <type 'numpy.bool_'>, 'Complex128': <type 'numpy.complex256'>, 'Complex32': <type 'numpy.complex64'>, 'Complex64': <type 'numpy.complex128'>, 'D': 'Complex64', 'Datetime64': <type 'numpy.datetime64'>, 'F': 'Complex32', 'Float128': <type 'numpy.float128'>, ...}
typecodes = {'All': '?bhilqpBHILQPefdgFDGSUVOMm', 'AllFloat': 'efdgFDG', 'AllInteger': 'bBhHillLqQpP', 'Character': 'c', 'Complex': 'FDG', 'Datetime': 'Mm', 'Float': 'efdg', 'Integer': 'bhilqp', 'UnsignedInteger': 'BHILQP'}
```