spectral\textsubscript{cube} Documentation

Release 0.4.5.dev1105

Adam Ginsburg, Tom Robitaille, Chris Beaumont, Adam Leroy, Erik Koch

Aug 01, 2019
I  Quick start  3
II  Using spectral-cube  7
The spectral-cube package provides an easy way to read, manipulate, analyze, and write data cubes with two positional dimensions and one spectral dimension, optionally with Stokes parameters. It provides the following main features:

• A uniform interface to spectral cubes, robust to the wide range of conventions of axis order, spatial projections, and spectral units that exist in the wild.

• Easy extraction of cube sub-regions using physical coordinates.

• Ability to easily create, combine, and apply masks to datasets.

• Basic summary statistic methods like moments and array aggregates.

• Designed to work with datasets too large to load into memory.
Part I

Quick start
Here's a simple script demonstrating the spectral-cube package:

```python
>>> import astropy.units as u
>>> from astropy.utils import data
>>> from spectral_cube import SpectralCube

>>> fn = data.get_pkg_data_filename('tests/data/adv.fits', 'spectral_cube')
>>> cube = SpectralCube.read(fn)
>>> print(cube)
SpectralCube with shape=(4, 3, 2) and unit=K:
  n_x: 2 type_x: RA---SIN unit_x: deg range: 24.062698 deg: 24.063349 deg
  n_y: 3 type_y: DEC--SIN unit_y: deg range: 29.934094 deg: 29.935209 deg
  n_s: 4 type_s: VOPT unit_s: km / s range: -321.215 km / s: -317.350 km / s

# extract the subcube between 98 and 100 GHz
>>> slab = cube.spectral_slab(98 * u.GHz, 100 * u.GHz)

# Ignore elements fainter than 1K
>>> masked_slab = slab.with_mask(slab > 1)

# Compute the first moment and write to file
>>> m1 = masked_slab.moment(order=1)
>>> m1.write('moment_1.fits')
```
Part II

Using spectral-cube
The package centers around the `SpectralCube` class. In the following sections, we look at how to read data into this class, manipulate spectral cubes, extract moment maps or subsets of spectral cubes, and write spectral cubes to files.
1.1 Installing spectral-cube

1.1.1 Requirements

This package has the following dependencies:

- Python 2.7 or later (Python 3.x is supported)
- Numpy 1.8 or later
- Astropy 1.0 or later
- radio_beam, used when reading in spectral cubes that use the BMAJ/BMIN convention for specifying the beam size.
- Bottleneck, optional (speeds up median and percentile operations on cubes with missing data)
- Regions >=0.3dev, optional (Serialises/Deserialises DS9/CRTF region files and handles them. Used when extracting a subcube from region)
- six

1.1.2 Installation

To install the latest stable release, you can type:

```
pip install spectral-cube
```

or you can download the latest tar file from PyPI and install it using:

```
python setup.py install
```

1.1.3 Developer version

If you want to install the latest developer version of the spectral cube code, you can do so from the git repository:

```
cd spectral-cube
python setup.py install
```

You may need to add the `--user` option to the last line if you do not have root access. You can also install the latest developer version in a single line with pip:

1.1.4 Installing into CASA

Installing packages in CASA is fairly straightforward. The process is described here. In short, you can do the follow-
ing:

First, we need to make sure pip is installed. Start up CASA as normal, and type:

```
CASA <1>: from setuptools.command import easy_install
CASA <2>: easy_install.main(['--user', 'pip'])
```

Now, quit CASA and re-open it, then type the following to install spectral-cube:

```
CASA <1>: import pip
CASA <2>: pip.main(['install', 'spectral-cube', '--user'])
```

1.2 Creating/reading spectral cubes

1.2.1 Importing

The `SpectralCube` class is used to represent 3-dimensional datasets (two positional dimensions and one spectral

dimension) with a World Coordinate System (WCS) projection that describes the mapping from pixel to world coor-
dinates and vice-versa. The class is imported with:

```
>>> from spectral_cube import SpectralCube
```

1.2.2 Reading from a file

In most cases, you are likely to read in an existing spectral cube from a file. The reader is designed to be able to deal

with any arbitrary axis order and always return a consistently oriented spectral cube (see Accessing data). To read in

a file, use the `read()` method as follows:

```
>>> cube = SpectralCube.read('L1448_13CO.fits')
```

This will always read the Stokes I parameter in the file. For information on accessing other Stokes parameters, see

stokes.

**Note:** In most cases, the FITS reader should be able to open the file in memory-mapped mode, which means that the
data is not immediately read, but is instead read as needed when data is accessed. This allows large files (including

larger than memory) to be accessed. However, note that certain FITS files cannot be opened in memory-mapped mode,
in particular compressed (e.g. .fits.gz) files. See the Handling large datasets page for more details about dealing

with large data sets.
1.2.3 Direct Initialization

If you are interested in directly creating a SpectralCube instance, you can do so using a 3-d Numpy-like array with a 3-d WCS object:

```python
>>> cube = SpectralCube(data=data, wcs=wcs)
```

Here data can be any Numpy-like array, including memory-mapped Numpy arrays (as mentioned in Reading from a file, memory-mapping is a technique that avoids reading the whole file into memory and instead accessing it from the disk as needed).

1.2.4 Hacks for simulated data

If you’re working on synthetic images or simulated data, where a location on the sky is not relevant (but the frequency/wavelength axis still is!), a hack is required to set up the world coordinate system. The header should be set up such that the projection is cartesian, i.e.:  

```python
CRVAL1 = 0
CTYPE1 = 'RA---CAR'
CRVAL2 = 0
CTYPE2 = 'DEC--CAR'
CDELT1 = 1.0e-4 //degrees
CDELT2 = 1.0e-4 //degrees
CUNIT1 = 'deg'
CUNIT2 = 'deg'
```

Note that the x/y axes must always have angular units (i.e., degrees). If your data are really in physical units, you should note that in the header in other comments, but spectral-cube doesn’t care about this.

If the frequency axis is irrelevant, spectral-cube is probably not the right tool to use; instead you should use astropy.io.fits or some other file reader directly.

1.3 Accessing data

Once you have initialized a SpectralCube() instance, either directly or by reading in a file, you can easily access the data values and the world coordinate information.

1.3.1 Data values

You can access the underlying data using the unmasked_data array which is a Numpy-like array:

```python
>>> slice_unmasked = cube.unmasked_data[0,:,::]
```

The order of the dimensions of the unmasked_data array is deterministic - it is always (n_spectral, n_y, n_x) irrespective of how the cube was stored on disk.

Note: The term unmasked indicates that the data is the raw original data from the file. SpectralCube() also allows masking of values, which is discussed in Masking.

If a slice is not specified, the object returned is not strictly a Numpy array, and will not work with all functions outside of the spectral_cube package that expect Numpy arrays. In order to extract a normal Numpy array, instead specify
a mask of [:] which will force the object to be converted to a Numpy array (the compulsory slicing is necessary in order to avoid memory-related issues with large data cubes).

### 1.3.2 World coordinates

Given a cube object, it is straightforward to find the coordinates along the spectral axis:

```
>>> cube.spectral_axis
[ -2.97198762e+03 -2.63992044e+03 -2.30785327e+03 -1.97578610e+03
  -1.64371893e+03 -1.31165176e+03 -9.79584583e+02 -6.47517411e+02
  ...  3.15629983e+04 3.18950655e+04 3.22271326e+04 3.25591998e+04
  3.28912670e+04 3.32233342e+04] m / s
```

The default units of a spectral axis are determined from the FITS header or WCS object used to initialize the cube, but it is also possible to change the spectral axis (see *Manipulating cubes and extracting subcubes*).

More generally, it is possible to extract the world coordinates of all the pixels using the `world` property, which returns the spectral axis then the two positional coordinates in reverse order (in the same order as the data indices).

```
>>> velo, dec, ra = cube.world[:]  # doctest: +SKIP
```

In order to extract coordinates, a slice (such as [:] above) is required. Using [:] will return three 3-d arrays with the coordinates for all pixels. Using e.g. `[0,:, :]` will return three 2-d arrays of coordinates for the first spectral slice.

If you forget to specify a slice, you will get the following error:

```
>>> velo, dec, ra = cube.world  # doctest: +SKIP
...
Exception: You need to specify a slice (e.g. `[:, :]` or `[:, :, :]`) in order to access this property.
```

In the case of large data cubes, requesting the coordinates of all pixels would likely be too slow, so the slicing allows you to compute only a subset of the pixel coordinates (see *Handling large datasets* for more information on dealing with large data cubes).

### 1.4 Masking

#### 1.4.1 Getting started

In addition to supporting the representation of data and associated WCS, it is also possible to attach a boolean mask to the `SpectralCube` class. Masks can take various forms, but one of the more common ones is a cube with the same dimensions as the data, and that contains e.g. the boolean value `True` where data should be used, and the value `False` when the data should be ignored (though it is also possible to flip the convention around; see *Inclusion and Exclusion*).

To create a boolean mask from a boolean array `mask_array`, you can for example use:

```
>>> from astropy import units as u
>>> from spectral_cube import BooleanArrayMask
>>> mask = BooleanArrayMask(mask=mask_array, wcs=cube.wcs)
```

**Note:** Currently, the mask convention is opposite of what is defined for Numpy masked array and Astropy Table.

Using a pure boolean array may not always be the most efficient solution, because it may require a large amount of memory.
You can also create a mask using simple conditions directly on the cube values themselves, for example:

```python
>>> mask = cube > 1.3*u.K
```

This is more efficient, because the condition is actually evaluated on-the-fly as needed. Note that units equivalent to the cube’s units must be used.

Masks can be combined using standard boolean comparison operators:

```python
>>> new_mask = (cube > 1.3*u.K) & (cube < 100.*u.K)
```

The available operators are & (and), | (or), and ~ (not).

To apply a new mask to a `SpectralCube` class, use the `with_mask()` method, which can take a mask and combine it with any pre-existing mask:

```python
>>> cube2 = cube.with_mask(new_mask)
```

In the above example, `cube2` contains a mask that is the & combination of `new_mask` with the existing mask on `cube`. The `cube2` object contains a view to the same data as `cube`, so no data is copied during this operation.

Boolean arrays can also be used as input to `with_mask()`, assuming the shape of the mask and the data match:

```python
>>> cube2 = cube.with_mask(boolean_array)
```

Any boolean area that can be broadcast to the cube shape can be used as a boolean array mask.

### 1.4.2 Accessing masked data

As mentioned in Accessing data, the raw and unmasked data can be accessed with the `unmasked_data` attribute. You can access the masked data using `filled_data`. This array is a copy of the original data with any masked value replaced by a fill value (which is `np.nan` by default but can be changed using the `fill_value` option in the `SpectralCube` initializer). The ‘filled’ data is accessed with e.g.:

```python
>>> slice_filled = cube.filled_data[0,:,:]
```

Note that accessing the filled data should still be efficient because the data are loaded and filled only once you access the actual data values, so this should still be efficient for large datasets.

If you are only interested in getting a flat (i.e. 1-d) array of all the non-masked values, you can also make use of the `flattened()` method:

```python
>>> flat_array = cube.flattened()
```

### 1.4.3 Fill values

When accessing the data (see Accessing data), the mask may be applied to the data and the masked values replaced by a fill value. This fill value can be set using the `fill_value` initializer in `SpectralCube`, and is set to `np.nan` by default. To change the fill value on a cube, you can make use of the `with_fill_value()` method:

```python
>>> cube2 = cube.with_fill_value(0.)
```

This returns a new `SpectralCube` instance that contains a view to the same data in `cube` (so no data are copied).
1.4.4 Inclusion and Exclusion

The term “mask” is often used to refer both to the act of excluding and including pixels from analysis. To be explicit about how they behave, all mask objects have an `include()` method that returns a boolean array. True values in this array indicate that the pixel is included/valid, and not filtered/replaced in any way. Conversely, True values in the output from `exclude()` indicate the pixel is excluded/invalid, and will be filled/filtered. The inclusion/exclusion behavior of any mask can be inverted via:

```python
>>> mask_inverse = ~mask
```

1.4.5 Advanced masking

Masks based on simple functions that operate on the initial data can be defined using the `LazyMask` class. The motivation behind the `LazyMask` class is that it is essentially equivalent to a boolean array, but the boolean values are computed on-the-fly as needed, meaning that the whole boolean array does not ever necessarily need to be computed or stored in memory, making it ideal for very large datasets. The function passed to `LazyMask` should be a simple function taking one argument - the dataset itself:

```python
>>> from spectral_cube import LazyMask
>>> cube = read(...)  
>>> LazyMask(np.isfinite, cube=cube)
```

or for example:

```python
>>> def threshold(data):
...   return data > 3.
>>> LazyMask(threshold, cube=cube)
```

As shown in *Getting Started*, `LazyMask` instances can also be defined directly by specifying conditions on `SpectralCube` objects:

```python
>>> cube > 5*u.K  # doctest: +SKIP
LazyComparisonMask(...)  
```

1.4.6 Outputting masks

The attached mask to the given `SpectralCube` class can be converted into a CASA image using `make_casa_mask()`:

```python
>>> from spectral_cube.io.casa_masks import make_casa_mask
>>> make_casa_mask(cube, 'casa_mask.image', add_stokes=False)  # doctest: +SKIP
```

Optionally, a redundant Stokes axis can be added to match the original CASA image.

**Note:** Outputting to CASA masks requires that `spectral_cube` be run from a CASA python session.

1.4.7 Masking cubes with other cubes

A common use case is to mask a cube based on another cube in the same coordinates. For example, you want to create a mask of 13CO based on the brightness of 12CO. This can be done straightforwardly if they are on an identical grid:
mask_12co = cube12co > 0.5*u.Jy
masked_cube13co = cube13co.with_mask(mask_12co)

If you see errors such as WCS does not match mask WCS, but you’re confident that your two cube are on the same grid, you should have a look at the cube.wcs attribute and see if there are subtle differences in the world coordinate parameters. These frequently occur when converting from frequency to velocity as there is inadequate precision in the rest frequency.

For example, these two axes are nearly identical, but not perfectly so:

Number of WCS axes: 3
CTYPE : 'RA---SIN' 'DEC--SIN' 'VRAD'
CRVAL : 269.08866286689999 -21.9562448137299999 -3000.000559989533
CRPIX : 161.0 161.0 1.0
PC1_1 PC1_2 PC1_3 : 1.0 0.0 0.0
PC2_1 PC2_2 PC2_3 : 0.0 1.0 0.0
PC3_1 PC3_2 PC3_3 : 0.0 0.0 1.0
CDELT : -1.3888888888888999e-05 1.3888888888889999e-05 299.99999994273281
NAXIS : 0 0

Number of WCS axes: 3
CTYPE : 'RA---SIN' 'DEC--SIN' 'VRAD'
CRVAL : 269.08866286689999 -21.9562448137299999 -3000.0000242346514
CRPIX : 161.0 161.0 1.0
PC1_1 PC1_2 PC1_3 : 1.0 0.0 0.0
PC2_1 PC2_2 PC2_3 : 0.0 1.0 0.0
PC3_1 PC3_2 PC3_3 : 0.0 0.0 1.0
CDELT : -1.3888888888888999e-05 1.3888888888889999e-05 300.00000001056611
NAXIS : 0 0

In order to compose masks from these, we need to set the wcs_tolerance parameter:

masked_cube13co = cube13co.with_mask(mask_12co, wcs_tolerance=1e-3)

which in this case will check equality at the 1e-3 level, which truncates the 3rd CRVAL to the point of equality before comparing the values.

1.5 Spectral Cube Arithmetic

Simple arithmetic operations between cubes and scalars, broadcastable numpy arrays, and other cubes are possible. However, such operations should be performed with caution because they require loading the whole cube into memory and will generally create a new cube in memory.

Examples:

import astropy.units as u
from astropy.utils import data
fn = data.get_pkg_data_filename('tests/data/adv.fits', 'spectral_cube')
from spectral_cube import SpectralCube
cube = SpectralCube.read(fn)
cube2 = cube * 2
cube3 = cube + 1.5*u.K
cube4 = cube2 + cube3

Each of these cubes is a new cube in memory. Note that for addition and subtraction, the units must be equivalent to those of the cube.
Please see *Handling large datasets* for details on how to perform arithmetic operations on a small subset of data at a time.

### 1.6 Manipulating cubes and extracting subcubes

#### 1.6.1 Modifying the spectral axis

As mentioned in *Accessing data*, it is straightforward to find the coordinates along the spectral axis using the `spectral_axis` attribute:

```python
>>> cube.spectral_axis
[ -2.97198762e+03 -2.63992044e+03 -2.30785327e+03 -1.97578610e+03
 -1.64371893e+03 -1.31165176e+03 -9.79584583e+02 -6.47517411e+02
 ... 3.15629983e+04 3.18950655e+04 3.22271326e+04 3.25591998e+04
 3.28912670e+04 3.32233342e+04] m / s
```

The default units of a spectral axis are determined from the FITS header or WCS object used to initialize the cube, but it is also possible to change the spectral axis unit using `with_spectral_unit()`:

```python
>>> from astropy import units as u

>>> cube2 = cube.with_spectral_unit(u.km / u.s)

>>> cube2.spectral_axis
[ -2.97198762e+00 -2.63992044e+00 -2.30785327e+00 -1.97578610e+00
 -1.64371893e+00 -1.31165176e+00 -9.79584583e-01 -6.47517411e-01
 ... 3.02347296e+01 3.05667968e+01 3.08988639e+01 3.12309311e+01
 3.15629983e+01 3.18950655e+01 3.22271326e+01 3.25591998e+01
 3.28912670e+01 3.32233342e+01] km / s
```

It is also possible to change from velocity to frequency for example, but this requires specifying the rest frequency or wavelength as well as a convention for the doppler shift calculation:

```python
>>> cube3 = cube.with_spectral_unit(u.GHz, velocity_convention='radio',
... rest_value=200 * u.GHz)

[ 220.40086492 220.40062079 220.40037667 220.40013254 220.39988841
 220.39964429 220.39940016 220.39915604 220.39891191 220.39866778
 ... 220.37498755 220.37474342 220.3744993 220.37425517] GHz
```

The new cubes will then preserve the new spectral units when computing moments for example (see *Moment maps and statistics*).

#### 1.6.2 Extracting a spectral slab

Given a spectral cube, it is easy to extract a sub-cube covering only a subset of the original range in the spectral axis. To do this, you can use the `spectral_slab()` method. This method takes lower and upper bounds for the spectral axis, as well as an optional rest frequency, and returns a new `SpectralCube` instance. The bounds can be specified as a frequency, wavelength, or a velocity but the units have to match the type of the spectral units in the cube (if they do not match, first use `with_spectral_unit()` to ensure that they are in the same units). The bounds should be given as Astropy `Quantities` as follows:
>>> from astropy import units as u
>>> subcube = cube.spectral_slab(-50 * u.km / u.s, +50 * u.km / u.s)

The resulting cube subcube (which is also a SpectralCube instance) then contains all channels that overlap with the range -50 to 50 km/s relative to the rest frequency assumed by the world coordinates, or the rest frequency specified by a prior call to `with_spectral_unit()`.

### 1.6.3 Extracting a sub-cube by indexing

It is also easy to extract a sub-cube from pixel coordinates using standard Numpy slicing notation:

```python
>>> sub_cube = cube[:100, 10:50, 10:50]
```

This returns a new SpectralCube object with updated WCS information.

### 1.6.4 Extracting a subcube from a DS9/CRTF region

You can use DS9/CRTF regions to extract subcubes. The minimal enclosing subcube will be extracted with a two-dimensional mask corresponding to the DS9/CRTF region. Regions is required for region parsing. CRTF regions may also contain spectral cutout information.

This example shows extraction of a subcube from a ds9 region file `file.reg`. `read_ds9` parses the ds9 file and converts it to a list of `Region` objects:

```python
>>> import regions
>>> region_list = regions.read_ds9("file.reg")
>>> sub_cube = cube.subcube_from_regions(region_list)
```

This one shows extraction of a subcube from a CRTF region file `file.crtf`, parsed using `read_crtf`:

```python
>>> import regions
>>> region_list = regions.read_crtf("file.crtf")
>>> sub_cube = cube.subcube_from_regions(region_list)
```

If you want to loop over individual regions with a single region file, you need to convert the individual regions to lists of that region:

```python
>>> region_list = regions.read_ds9("file.reg")
>>> for region in region_list:
...     sub_cube = cube.subcube_from_regions([region])
```

You can also directly use a ds9 region string. This example extracts a 0.1 degree circle around the Galactic Center:

```python
>>> region_str = "galactic; circle(0, 0, 0.1)"
>>> sub_cube = cube.subcube_from_ds9region(region_str)
```

Similarly, you can also use a CRTF region string:

```python
>>> region_str = "circle[[0deg, 0deg], 0.1deg], coord=galactic, range=[150km/s, 300km/s]"
>>> sub_cube = cube.subcube_from_crtfregion(region_str)
```

CRTF regions that specify a subset in the spectral dimension can be used to produce full 3D cutouts. The meta attribute of a `regions.Region` object contains the spectral information for that region in the three special keywords `range`, `restfreq`, and `veltype`.

### 1.6. Manipulating cubes and extracting subcubes 19
>>> import regions
>>> from astropy import units as u

>>> regpix = regions.RectanglePixelRegion(regions.PixCoord(0.5, 1), width=4, height=2)
>>> regpix.meta['range'] = [150 * u.km/u.s, 300 * u.km/u.s] # spectral range
>>> regpix.meta['restfreq'] = [100 * u.GHz] # rest frequency
>>> regpix.meta['veltype'] = 'OPTICAL' # velocity convention
>>> subcube = cube.subcube_from_regions([regpix])

If range is specified, but the other two keywords are not, the code will likely crash.

### 1.6.5 Extract the minimal valid subcube

If you have a mask that masks out some of the cube edges, such that the resulting sub-cube might be smaller in memory, it can be useful to extract the minimal enclosing sub-cube:

```python
>>> sub_cube = cube.minimal_subcube()
```

You can also shrink any cube by this mechanism:

```python
>>> sub_cube = cube.with_mask(smaller_region).minimal_subcube()
```

### 1.6.6 Extract a spatial and spectral subcube

There is a generic subcube function that allows slices in the spatial and spectral axes simultaneously, as long as the spatial axes are aligned with the pixel axes. An arbitrary example looks like this:

```python
>>> sub_cube = cube.subcube(xlo=5*u.deg, xhi=6*u.deg,
                               ylo=2*u.deg, yhi=2.1*u.deg,
                               zlo=50*u.GHz, zhi=51*u.GHz)
```

### 1.7 Metadata and Headers

The metadata of both `SpectralCube` s and `LowerDimensionalObject` s is stored in their `.meta` attribute, which is a dictionary of metadata.

When writing these objects to file, or exporting them as FITS HDUs, the metadata will be written to the FITS header. If the metadata matches the FITS standard, it will just be directly written, with the dictionary keys replaced with uppercase versions. If the keys are longer than 8 characters, a FITS `COMMENT` entry will be entered with the data formatted as `{key}={value}`.

The world coordinate system (WCS) metadata will be handled automatically, as will the beam parameter metadata. The automation implies that WCS keywords and beam keywords cannot be manipulated directly by changing the meta dictionary; they must be manipulated through other means (e.g., `Manipulating cubes and extracting subcubes`).

### 1.8 Smoothing

There are two types of smoothing routine available in `spectral_cube`: spectral and spatial.
1.8.1 Spatial Smoothing

The `convolve_to` method will convolve each plane of the cube to a common resolution, assuming the cube’s resolution is known in advance and stored in the cube’s `beam` or `beams` attribute.

A simple example:

```python
import radio_beam
from spectral_cube import SpectralCube
from astropy import units as u

cube = SpectralCube.read('file.fits')
beam = radio_beam.Beam(major=1*u.arcsec, minor=1*u.arcsec, pa=0*u.deg)
new_cube = cube.convolve_to(beam)
```

Note that the `convolve_to()` method will work for both `VaryingResolutionSpectralCube` instances and single-resolution `SpectralCube` instances, but for a `VaryingResolutionSpectralCube`, the convolution kernel will be different for each slice.

**Common Beam selection**

You may want to convolve your cube to the smallest beam that is still larger than all contained beams. To do this, you can use the `common_beam` tool. For example:

```python
common_beam = cube.beams.common_beam()
new_cube = cube.convolve_to(common_beam)
```

Sometimes, you’ll encounter the error “Could not find common beam to deconvolve all beams.” This is a real issue, as the algorithms we have in hand so far do not always converge on a common containing beam. There are two ways to get the algorithm to converge to a valid common beam:

1. **Changing the tolerance.** - You can try to change the tolerance used in the `getMinVolEllipse` code by passing `tolerance=1e-5` to the common beam function:

   ```python
cube.beams.common_beam(tolerance=1e-5)
   ```

   Convergence may be met by either increasing or decreasing the tolerance; it depends on having the algorithm not step within the minimum enclosing ellipse, leading to the error. Note that decreasing the tolerance by an order of magnitude will require an order of magnitude more iterations for the algorithm to converge and will take longer to run.

2. **Changing epsilon** - A second parameter `epsilon` controls the fraction to overestimate the beam size, ensuring that solutions that are marginally smaller than the common beam will not be found by the algorithm:

   ```python
cube.beams.common_beam(epsilon=1e-3)
   ```

   The default value of `epsilon=1e-3` will sample points 0.1% larger than the edge of each beam in the set. Increasing `epsilon` ensures that a valid common beam can be found, avoiding the tolerance issue, but will result in overestimating the common beam area. For most radio data sets, where the beam is oversampled by $\sim 5$ pixels, moderate increases in `epsilon` will increase the common beam area far less than a pixel area, making the overestimation negligible.

   We recommend testing different values of tolerance to find convergence, and if the error persists, to then slowly increase `epsilon` until a valid common beam is found. More information can be found in the [radio-beam documentation](https://examples.readthedocs.io/en/latest/).
1.8.2 Spectral Smoothing

Only SpectralCube instances with a consistent beam can be spectrally smoothed, so if you have a VaryingResolutionSpectralCube, you must convolve each slice in it to a common resolution before spectrally smoothing. spectral_smooth() will apply a convolution kernel to each spectrum in turn. As of July 2016, a parallelized version is partly written but incomplete.

Example:

```python
import radio_beam
from spectral_cube import SpectralCube
from astropy import units as u
from astropy.convolution import Gaussian1DKernel

cube = SpectralCube.read('file.fits')
kern = Gaussian1DKernel(2.5)
new_cube = cube.spectral_smooth(kern)
```

This can be useful if you want to interpolate onto a coarser grid but maintain Nyquist sampling. You can then use the spectral_interpolate method to regrid your smoothed spectrum onto a new grid.

Say, for example, you have a cube with 0.5 km/s resolution, but you want to resample it onto a 2 km/s grid. You might then choose to smooth by a factor of 4, then downsample by the same factor:

```python
# cube.spectral_axis is np.arange(0,10,0.5) for this example
new_axis = np.arange(0,10,2)*u.km/u.s
fwhm_factor = np.sqrt(8*np.log(2))
smcube = cube.spectral_smooth(Gaussian1DKernel(4/fwhm_factor))
interp_Cube = smcube.spectral_interpolate(new_axis,
                                          suppress_smooth_warning=True)
```

We include the suppress_smooth_warning override because there is no way for SpectralCube to know if you've done the appropriate smoothing (i.e., making sure that your new grid nyquist samples the data) prior to the interpolation step. If you don’t specify this, it will still work, but you’ll be warned that you should preserve Nyquist sampling.

If you have a cube with 0.1 km/s resolution (where we assume resolution corresponds to the fwhm of a gaussian), and you want to smooth it to 0.25 km/s resolution, you can smooth the cube with a Gaussian Kernel that has a width of \((0.25^2 - 0.1^2)^{0.5} = 0.229 \text{ km/s}\). For simplicity, it can be done in the unit of pixel. In our example, each channel is 0.1 km/s wide:

```python
import numpy as np
from astropy import units as u
from spectral_cube import SpectralCube
from astropy.convolution import Gaussian1DKernel

cube = SpectralCube.read('file.fits')
fwhm_factor = np.sqrt(8*np.log(2))
current_resolution = 0.1 * u.km/u.s
target_resolution = 0.25 * u.km/u.s
pixel_scale = 0.1 * u.km/u.s
gaussian_width = ((target_resolution**2 - current_resolution**2)**0.5 / pixel_scale) / fwhm_factor
kernel = Gaussian1DKernel(gaussian_width)
new_cube = cube.spectral_smooth(kernel)
new_cube.write('newfile.fits')
```
1.9 Writing spectral cubes

You can write out a SpectralCube instance by making use of the write() method:

```python
>>> cube.write('new_cube.fits', format='fits')
```

1.10 Moment maps and statistics

1.10.1 Moment maps

Producing moment maps from a SpectralCube instance is straightforward:

```python
>>> moment_0 = cube.moment(order=0)
>>> moment_1 = cube.moment(order=1)
>>> moment_2 = cube.moment(order=2)
```

By default, moments are computed along the spectral dimension, but it is also possible to pass the axis argument to compute them along a different axis:

```python
>>> moment_0_along_x = cube.moment(order=0, axis=2)
```

Note: These follow the mathematical definition of moments, so the second moment is computed as the variance. For the actual formulas used for the moments, please see the relevant documentation here. For linewidth maps, see the Linewidth maps section.

You may also want to convert the unit of the datacube into a velocity one before you can obtain a genuine velocity map via a 1st moment map. So first it will be necessary to apply the with_spectral_unit method from this package with the proper attribute settings:

```python
>>> nii_cube = cube.with_spectral_unit(u.km/u.s, velocity_convention='optical', rest_value=6584*u.AA)
```

Note that the rest_value in the above code refers to the wavelength of the targeted line in the 1D spectrum corresponding to the 3rd dimension. Also, since not all velocity values are relevant, next we will use the spectral_slab method to slice out the chunk of the cube that actually contains the line:

```python
>>> nii_cube = cube.with_spectral_unit(u.km/u.s, velocity_convention='optical', rest_value=6584*u.AA).spectral_slab(-60*u.km/u.s, -20*u.km/u.s)
```

Finally, we can now generate the 1st moment map containing the expected velocity structure:

```python
>>> moment_1 = nii_cube.moment(order=1)
```

The moment maps returned are Projection instances, which act like Quantity objects, and also have convenience methods for writing to a file:

```python
>>> moment_0.write('moment0.fits')
>>> moment_1.write('moment1.fits')
```

and converting the data and WCS to a FITS HDU:

```python
>>> moment_0.hdu
<astropy.io.fits.hdu.image.PrimaryHDU at 0x10d6ec510>
```
The conversion to HDU objects makes it very easy to use the moment map with plotting packages such as APL.py:

```python
>>> import aplpy
>>> f = aplpy.FITSFigure(moment_0.hdu)
>>> f.show_colorscale()
>>> f.save('moment_0.png')
```

### 1.10.2 Linewidth maps

Making linewidth maps (sometimes referred to as second moment maps in radio astronomy), you can use:

```python
>>> sigma_map = cube.linewidth_sigma()
>>> fwhm_map = cube.linewidth_fwhm()
```

These also return `Projection` instances as for the Moment maps.

### 1.11 Explanations of commonly-encountered error messages

#### 1.11.1 Beam parameters differ

If you are using spectral cubes produced by CASA's tclean, it may have a different beam size for each channel. In this case, it will be loaded as a `VaryingResolutionSpectralCube` object. If you perform any operations spanning the spectral axis, for example `cube.moment0(axis=0)` or `cube.max(axis=0)`, you may encounter errors like this one:

*Beam srs differ by up to 1.0x, which is greater than the threshold 0.01.*

This occurs if the beam sizes are different by more than the specified threshold factor. A default threshold of 1% is set because for most interferometric images, beam differences on this scale are negligible (they correspond to flux measurement errors of 10^-4).

To inspect the beam properties, look at the `beams` attribute, for example:

```python
>>> cube.beams
[Beam: BMAJ=1.1972888768114624 arcsec BMIN=1.0741511583328247 arcsec BPA=72.7129635009766 deg,
 Beam: BMAJ=1.1972869634628296 arcsec BMIN=1.0741279125213623 arcsec BPA=72.71561431884766 deg,
 Beam: BMAJ=1.197291702529907 arcsec BMIN=1.0741302967071533 arcsec BPA=72.71575164794922 deg,
 ...]
```

In this example, the beams differ by a tiny amount that is below the threshold. However, sometimes you will encounter cubes with dramatically different beam sizes, and spectral-cube will prevent you from performing operations along the spectral axis with these beams because such operations are poorly defined.

There are several options to manage this problem:

1. Increase the threshold. This is best done if the beams still differ by a small amount, but larger than 1%. To do this, set `cube.beam_threshold = [new value]`. This is the “tape over the check engine light” approach; use with caution.

2. Convolve the cube to a common resolution using `convolve_to`. This is again best if the largest beam is only slightly larger than the smallest.

3. Mask out the bad channels. For example:
good_beams = cube.identify_bad_beams(threshold=0.1)
mcube = cube.mask_out_bad_beams(threshold=0.1)

### 1.11.2 Moment-2 or FWHM calculations give unexpected NaNs

It is fairly common to have moment 2 calculations return NaN values along pixels where real values are expected, e.g., along pixels where both moment0 and moment1 return real values.

Most commonly, this is caused by “bad baselines”, specifically, by large sections of the spectrum being slightly negative at large distances from the centroid position (the moment 1 position). Because moment 2 weights pixels at larger distances more highly (as the square of the distance), slight negative values at large distances can result in negative values entering the square root when computing the line width or the FWHM.

The solution is either to make a tighter mask, excluding the pixels far from the centroid position, or to ensure that the baseline does not have any negative systematic offset.

### 1.11.3 Looking at images with matplotlib

Matplotlib accesses a lot of hidden properties of arrays when plotting. If you try to show a slice with imshow, you may encounter the WCS-related error:

```
NotImplementedError: Reversing an axis is not implemented.
```

If you see this error, the only solution at present is to specify `origin='lower'`, which is the standard for images anyway. For example:

```python
import pylab as pl
pl.imshow(cube[5,:,:], origin='lower')
```

should work, where `origin='upper'` will not. This is due to a limitation in astropy.wcs slicing.

An alternative option, if it is absolutely necessary to use `origin='upper'` or if you encounter other matplotlib-related issues, is to use the `.value` attribute of the slice to get a bald numpy array to plot:

```python
import pylab as pl
pl.imshow(cube[5,:,:].value)
```

### 1.11.4 Silencing Warnings

If you don’t like seeing warnings about potential slowdowns, etc., the following will catch and disable those warnings (see also http://docs.astropy.org/en/stable/warnings.html):

```python
import warnings
from spectral_cube.utils import SpectralCubeWarning
warnings.filterwarnings(action='ignore', category=SpectralCubeWarning, append=True)
```

This will prevent any spectral-cube related warnings from being displayed. If you’d like more granular control over which warnings to ignore, look at spectral-cube/utils.py, which lists a wide range of warning types.
1.12 Quick Looks

Once you’ve loaded a cube, you inevitably will want to look at it in various ways. Slices in any direction have quicklook methods:

```python
>>> cube[50,:,:].quicklook()  # show an image  # doctest: +SKIP
>>> cube[:, 50, 50].quicklook()  # plot a spectrum  # doctest: +SKIP
```

The same can be done with moments:

```python
>>> cube.moment0(axis=0).quicklook()  # doctest: +SKIP
```

1.12.1 PVSlicer

The pveXtractor package comes with a GUI that has a simple matplotlib image viewer. To activate it for your cube:

```python
>>> cube.to_pveXtractor()  # doctest: +SKIP
```

1.13 Handling Beams

If you are using radio data, your cubes should have some sort of beam information included. spectral-cube handles beams using the radio_beam package.

There are two ways beams can be stored in FITS files: as FITS header keywords (BMAJ, BMIN, and BPA) or as a BinTableHDU extension. If the latter is present, spectral-cube will return a VaryingResolutionSpectralCube object.

For the simpler case of a single beam across all channels, the presence of the beam allows for direct conversion of a cube with Jy/beam units to surface brightness (K) units. Note, however, that this requires loading the entire cube into memory!

```python
>>> cube.unit
Unit("Jy / beam")
>>> kcube = cube.to(u.K)
>>> kcube.unit
Unit("K")
```

1.13.1 Adding a Beam

If your cube does not have a beam, a custom beam can be attached given:

```python
>>> new_beam = Beam(1. * u.deg)
>>> new_cube = cube.with_beam(new_beam)
>>> new_cube.beam
Beam: BMAJ=3600.0 arcsec BMIN=3600.0 arcsec BPA=0.0 deg
```

This is handy for synthetic observations, which initially have a point-like beam:

```python
>>> point_beam = Beam(0.0 * u.deg)
>>> new_cube = synth_cube.with_beam(point_beam)
>>> new_cube.beam
Beam: BMAJ=0.0 arcsec BMIN=0.0 arcsec BPA=0.0 deg
```
The cube can then be convolved to a new resolution:

```python
>>> new_beam = Beam(60 * u.arcsec)
>>> conv_synth_cube = synth_cube.convolve_to(new_beam)
>>> conv_synth_cube.beam
Beam: BMAJ=60.0 arcsec BMIN=60.0 arcsec BPA=0.0 deg
```

Beam can also be attached in the same way for `Projection` and `Slice` objects.

### 1.13.2 Multi-beam cubes

Varying resolution (multi-beam) cubes are somewhat trickier to work with in general, though unit conversion is easy. You can perform the same sort of unit conversion with `VaryingResolutionSpectralCube`s as with regular `SpectralCube`s; `spectral-cube` will use a different beam and frequency for each plane.

You can identify channels with bad beams (i.e., beams that differ from a reference beam, which by default is the median beam) using `identify_bad_beams` (the returned value is a mask array where `True` means the channel is good), mask channels with undesirable beams using `mask_out_bad_beams`, and in general mask out individual channels using `mask_channels`.

For other sorts of operations, discussion of how to deal with these cubes via smoothing to a common resolution is in the `Smoothing` document.

### 1.14 Spectral Extraction

A commonly required operation is extracting a spectrum from a part of a cube.

The simplest way to get a spectrum from the cube is simply to slice it along a single pixel:

```python
>>> spectrum = cube[:, 50, 60]
```

Slicing along the first dimension will create a `OneDSpectrum` object, which has a few useful capabilities.

#### 1.14.1 Aperture Extraction

Going one level further, you can extract a spectrum from an aperture. We’ll start with the simplest variant: a square aperture. The cube can be sliced in pixel coordinates to produce a sub-cube which we then average spatially to get the spectrum:

```python
>>> subcube = cube[:, 50:53, 60:63]
>>> spectrum = subcube.mean(axis=(1,2))
```

The spectrum can be obtained using any mathematical operation, such as the `max` or `std`, e.g., if you wanted to obtain the noise spectrum.

#### 1.14.2 Slightly more sophisticated aperture extraction

To get the flux in a circular aperture, you need to mask the data. In this example, we don’t use any external libraries, but show how to create a circular mask from scratch and apply it to the data:
1.14.3 Aperture and spectral extraction using regions

Spectral-cube supports ds9 and crtf regions, so you can use them to create a mask. The ds9/crtf region support relies on regions, which supports most shapes in ds9 and crtf, so you are not limited to circular apertures.

In this example, we’ll extract a subcube from ds9 region string using subcube_from_ds9region:

```python
>>> ds9_str = 'fk5; circle(19:23:43.907, +14:30:34.66, 3")'
>>> subcube = cube.subcube_from_ds9region(ds9_str)
>>> spectrum = subcube.mean(axis=(1, 2))
```

Similarly, one can extract a subcube from a crtf region string using subcube_from_crtfregion:

```python
>>> crtf_str = 'circle[[19:23:43.907, +14:30:34.66], 3", coord=fk5, range=[150km/s, 300km/s]]'
>>> subcube = cube.subcube_from_crtfregion(crtf_str)
>>> spectrum = subcube.mean(axis=(1, 2))
```

You can also use a list of Region objects to extract a subcube using subcube_from_regions:

```python
>>> import regions
>>> regpix = regions.RectanglePixelRegion(regions.PixCoord(0.5, 1), width=4, height=2)
>>> subcube = cube.subcube_from_regions([regpix])
>>> spectrum = subcube.mean(axis=(1, 2))
```

To learn more, go to Extracting a subcube from a DS9/CRTF region.

1.15 Continuum Subtraction

A common task with data cubes is continuum identification and subtraction. For line-rich cubes where the continuum is difficult to identify, you should use statcont. For single-line cubes, the process is much easier.

First, the simplest case is when you have a single line that makes up a small fraction of the total observed band, e.g., a narrow line. In this case, you can use a simple median approximation for the continuum:

```python
>>> med = cube.median(axis=0)
>>> med_sub_cube = cube - med
```

The second part of this task may complain that the cube is too big. If it does, you can still do the above operation by first setting `cube.allow_huge_operations=True`, but be warned that this can be expensive.

For a more complicated case, you may want to mask out the line-containing channels. This can be done using a spectral boolean mask:

```python
>>> from astropy import units as u
>>> import numpy as np
>>> spectral_axis = cube.with_spectral_unit(u.km/u.s).spectral_axis
>>> good_channels = (spectral_axis < 25*u.km/u.s) | (spectral_axis > 45*u.km/u.s)
```
The array `good_channels` is a simple 1D numpy boolean array that is True for all channels below 25 km/s and above 45 km/s, and is False for all channels in the range 25-45 km/s. The indexing trick `good_channels[:, np.newaxis, np.newaxis]` (or equivalently, `good_channels[:, None, None]`) is just a way to tell the cube which axes to project along. In more recent versions of `spectral-cube`, the indexing trick is not necessary. The median in this case is computed only over the specified line-free channels.

Any operation can be used to compute the continuum, such as the mean or some percentile, but for most use cases, the median is fine.

### 1.16 Examples

Note that these examples are not tested by continuous integration tests; it is possible they will become out-of-date over time. If you notice any mistakes or inconsistencies, please post them at https://github.com/radio-astro-tools/spectral-cube/issues.

1. From a cube with many lines, extract each line and create moment maps using the brightest line as a mask:

```python
>>> masked_cube = cube.with_mask(good_channels[:, np.newaxis, np.newaxis])
>>> med = masked_cube.median(axis=0)
>>> med_sub_cube = cube - med
```

The array `good_channels` is a simple 1D numpy boolean array that is True for all channels below 25 km/s and above 45 km/s, and is False for all channels in the range 25-45 km/s. The indexing trick `good_channels[:, np.newaxis, np.newaxis]` (or equivalently, `good_channels[:, None, None]`) is just a way to tell the cube which axes to project along. In more recent versions of `spectral-cube`, the indexing trick is not necessary. The median in this case is computed only over the specified line-free channels.

Any operation can be used to compute the continuum, such as the mean or some percentile, but for most use cases, the median is fine.

spectral_cube Documentation, Release 0.4.5.dev1105

(continued from previous page)

velocity_convention='optical')
    .spectral_slab(vz-width, vz+width))

# velocity of the brightest pixel
peak_velocity = brightest_cube.spectral_axis[brightest_cube.argmax(axis=0)]

# make a spatial mask excluding pixels with no signal
peak_amplitude = brightest_cube.max(axis=0)

# Create a noise map from a line-free region.  
# found this range from inspection of a spectrum:
# s = cube.max(axis=(1,2))
# s.quicklook()
noisemap = cube.spectral_slab(362.603*u.GHz, 363.283*u.GHz).std(axis=0)

spatial_mask = peak_amplitude > 3*noisemap

# Now loop over EACH line, extracting moments etc. from the appropriate region:  
# we'll also apply a transition-dependent width (my_line_widths) here because  
# these fainter lines do not have peaks as far out as the bright line.
for line_name, line_freq, line_width in zip(my_line_names, my_line_list, my_line_widths):
    subcube = cube.with_spectral_unit(u.km/u.s,  
                                       rest_value=line_freq,  
                                       velocity_convention='optical')
    .spectral_slab(peak_velocity.min()-line_width,  
                   peak_velocity.max()+line_width)

    # this part makes a cube of velocities for masking work
    temp = subcube.spectral_axis
    velocities = np.tile(temp[:,None,None], subcube.shape[1:])
    # `velocities` has the same shape as `subcube`

    # now we use the velocities from the brightest line to create a mask region
    # in the same velocity range but with different rest frequencies (different
    # lines)
    mask = np.abs(peak_velocity - velocities) < line_width

    # Mask on a pixel-by-pixel basis with a 1-sigma cut
    signal_mask = subcube > noisemap

    # the mask is a cube, the spatial mask is a 2d array, but in this case
    # numpy knows how to combine them properly
    # (signal_mask is a different type, so it can't be combined with the others
    msubcube = subcube.with_mask(mask & spatial_mask).with_mask(signal_mask)

    # Then make & save the moment maps
    for moment in (0,1,2):
        mom = msubcube.moment(order=moment, axis=0)
        mom.hdu.writeto("moment{0}_{1}_{2}_moment{0}.fits".format(moment, target, line_name),  
                        clobber=True)

2. Use aplpy (in a slightly unsupported way) to make an RGB velocity movie

```python
import aplpy
```
cube = SpectralCube.read('file.fits')
prefix = 'HC3N'

# chop out the NaN borders
cmin = cube.minimal_subcube()

# Create the WCS template
F = aplpy.FITSFigure(cmin[0].hdu)

# decide on the velocity range
v1 = 30*u.km/u.s
v2 = 60*u.km/u.s

# determine pixel range
p1 = cmin.closest_spectral_channel(v1)
p2 = cmin.closest_spectral_channel(v2)

for jj,ii in enumerate(range(p1, p2-1)):
    rgb = np.array([cmin[ii+2], cmin[ii+1], cmin[ii]]).T.swapaxes(0,1)
    # in case you manually set min/max
    rgb[rgb > max.value] = 1
    rgb[rgb < min.value] = 0
    # this is the unsupported little bit...
    F._ax1.clear()
    F._ax1.imshow((rgb-min.value)/(max-min).value, extent=F._extent)
    v1_ = int(np.round(cube.spectral_axis[ii].value))
    v2_ = int(np.round(cube.spectral_axis[ii+2].value))
    # then write out the files
    F.save('rgb/{2}_v{0}to{1}.png'.format(v1_, v2_, prefix))
    # make a sorted version for use with ffmpeg
    os.remove('rgb/{0:04d}.png'.format(jj))
    os.link('rgb/{2}_v{0}to{1}.png'.format(v1_, v2_, prefix), 'rgb/{0:04d}.png'.format(jj))
    print("Done with frame {1}: channel {0}".format(ii, jj))
    os.system('ffmpeg -y -i rgb/%04d.png -c:v libx264 -pix_fmt yuv420p -vf "scale=1024:768,setpts=10*PTS" -r 10 rgb/{0}_RGB_movie.mp4'.format(prefix))

3. Extract a beam-weighted spectrum from a cube

Each spectral cube has a 'beam' parameter if you have radio_beam installed. You can use that to create a beam kernel:

```python
kernel = cube.beam.as_kernel(cube.wcs.pixel_scale_matrix[1,1])
```

Find the pixel you want to integrate over form the image. e.g.,

```python
x,y = 500, 150
```

Then, cut out an appropriate sub-cube and integrate over it

```python
kernsize = kernel.shape[0]
subcube = cube[:y-kernsize/2.:y+kernsize/2., x-kernsize/2.:x+kernsize/2.]
# create a boolean mask at the 1% of peak level (you can modify this)
```
mask = kernel.array > (0.01*kernel.array.max())
msubcube = subcube.with_mask(mask)
# Then, take an appropriate beam weighting
weighted_cube = msubcube * kernel.array
# and *sum* (do not average!) over the weighted cube.
beam_weighted_spectrum = weighted_cube.sum(axis=(1,2))

1.17 Visualization

Spectral-cube is not primarily a visualization package, but it has several tools for visualizing subsets of the data.

All lower-dimensional subsets, OneDSpectrum, and Projection, have their own quicklook methods (quicklook and quicklook, respectively). These methods will plot the data with somewhat properly labeled axes.

The two-dimensional viewers default to using aplpy. Because of quirks of how aplpy sets up its plotting window, these methods will create their own figures. If use_aplpy is set to False, and similarly if you use the OneDSpectrum quicklook, the data will be overplotted in the latest used plot window.

In principle, one can also simply plot the data. For example, if you have a cube, you could do:

```python
>>> plt.plot(cube[:,0,0])
```
to plot a spectrum sliced out of the cube or:

```python
>>> plt.imshow(cube[0,:,0])
```
to plot an image slice.

**Warning:** There are known incompatibilities with the above plotting approach: matplotlib versions <2.1 will crash, and you will have to clear the plot window to reset it.

1.18 Other Visualization Tools

To visualize the cubes directly, you can use some of the other tools we provide for pushing cube data into external viewers.

See [Visualizing spectral cubes with yt](#) for using yt as a visualization tool.

The spectral_cube.SpectralCube.to_glue and spectral_cube.SpectralCube.to_ds9 methods will send the whole cube to glue and ds9. This approach generally requires loading the whole cube into memory.

There is also an astropy tutorial on accessing and manipulating FITS cubes with spectral-cube.
2.1 Visualizing spectral cubes with yt

2.1.1 Extracting yt objects

The SpectralCube class includes a `to_yt()` method that makes it easy to return an object that can be used by yt to make volume renderings or other visualizations of the data. One common issue with volume rendering of spectral cubes is that you may not want pixels along the spectral axis to be given the same ‘3-d’ size as positional pixels, so the `to_yt()` method includes a `spectral_factor` argument that can be used to compress or expand the spectral axis.

The `to_yt()` method is used as follows:

```python
>>> ytcube = cube.to_yt(spectral_factor=0.5)
>>> ds = ytcube.dataset
```

Warning: The API change in https://github.com/radio-astro-tools/spectral-cube/pull/129 affects the interpretation of the 0-pixel. There may be a 1-pixel offset between the yt cube and the SpectralCube

The `ds` object is then a yt object that can be used for rendering! By default the dataset is defined in pixel coordinates, going from \(0.5\) to \(n+0.5\), as would be the case in ds9, for example. Along the spectral axis, this range will be modified if `spectral_factor` does not equal unity.

When working with datasets in yt, it may be useful to convert world coordinates to pixel coordinates, so that whenever you may have to input a position in yt (e.g., for slicing or volume rendering) you can get the pixel coordinate that corresponds to the desired world coordinate. For this purpose, the method `world2yt()` is provided:

```python
>>> import astropy.units as u
>>> pix_coord = ytcube.world2yt([51.424522,
...                          30.723611,
...                          5205.18071], # units of deg, deg, m/s
...                          )
```

There is also a reverse method provided, `yt2world()`:

```python
>>> world_coord = ytcube.yt2world([ds.domain_center])
```

which in this case would return the world coordinates of the center of the dataset in yt.

Note: The `to_yt()` method and its associated coordinate methods are compatible with both yt v. 2.x and v. 3.0 and following, but use of version 3.0 or later is recommended due to substantial improvements in support for FITS data.
For more information on how yt handles FITS datasets, see the yt docs.

### 2.1.2 Visualization example

This section shows an example of a rendering script that can be used to produce a 3-d isocontour visualization using an object returned by `to_yt()`:

```python
import numpy as np
from spectral_cube import SpectralCube
from yt.mods import ColorTransferFunction, write_bitmap
import astropy.units as u

# Read in spectral cube
cube = SpectralCube.read('/quotesingle.VarL1448_13CO.fits', format='fits')

# Extract the yt object from the SpectralCube instance
ytcube = cube.to_yt(spectral_factor=0.75)
ds = ytcube.dataset

# Set the number of levels, the minimum and maximum level and the width of the isocontours
n_v = 10
vmin = 0.05
vmax = 4.0
dv = 0.02

# Set up color transfer function
transfer = ColorTransferFunction((vmin, vmax))
transfer.add_layers(n_v, dv, colormap='RdBu_r')

# Set up the camera parameters
# Derive the pixel coordinate of the desired center
# from the corresponding world coordinate
center = ytcube.world2yt([[51.424522, 30.723611, 5205.18071]])
direction = np.array([1.0, 0.0, 0.0])
width = 100. # pixels
size = 1024

camera = ds.camera(center, direction, width, size, transfer,
fields=['flux'])

# Take a snapshot and save to a file
snapshot = camera.snapshot()
write_bitmap(snapshot, 'cube_rendering.png', transpose=True)
```

You can move the camera around; see the yt camera docs.

### 2.1.3 Movie Making

There is a simple utility for quick movie making. The default movie is a rotation of the cube around one of the spatial axes, going from PP -> PV space and back.
>>> cube = read('cube.fits', format='fits')
>>> ytcube = cube.to_yt()
>>> images = ytcube.quick_render_movie('outdir')

The movie only does rotation, but it is a useful stepping-stone if you wish to learn how to use yt's rendering system.

Example:

### 2.1.4 SketchFab Isosurface Contours

For data exploration, making movies can be tedious - it is difficult to control the camera and expensive to generate new renderings. Instead, creating a 'model' from the data and exporting that to SketchFab can be very useful. Only grayscale figures will be created with the quicklook code.

You need an account on sketchfab.com for this to work.:

```python
>>> ytcube.quick_isocontour(title='GRS l=49 T3CO 1 K contours', level=1.0)
```

Here's an example:

You can also export locally to .ply and .obj files, which can be read by many programs (sketchfab, meshlab, blender). See the yt page for details.:

```python
>>> ytcube.quick_isocontour(export_to='ply', filename='meshes.ply', level=1.0)
>>> ytcube.quick_isocontour(export_to='obj', filename='meshes', level=1.0)
```

### 2.2 Handling large datasets

The `SpectralCube` class is designed to allow working with files larger than can be stored in memory. To take advantage of this and work effectively with large spectral cubes, you should keep the following three ideas in mind:

- Work with small subsets of data at a time.
- Minimize data copying.
- Minimize the number of passes over the data.

#### 2.2.1 Work with small subsets of data at a time

Numpy supports a *memory-mapping* mode which means that the data is stored on disk and the array elements are only loaded into memory when needed. `spectral_cube` takes advantage of this if possible, to avoid loading large files into memory.

Typically, working with NumPy involves writing code that operates on an entire array at once. For example:

```python
x = <a numpy array>
y = np.sum(np.abs(x * 3 + 10), axis=0)
```

Unfortunately, this code creates several temporary arrays whose size is equal to `x`. This is infeasible if `x` is a large memory-mapped array, because an operation like `(x * 3)` will require more RAM than exists on your system. A better way to compute `y` is by working with a single slice of `x` at a time:
```
y = np.zeros_like(x[0])
for plane in x:
    y += np.abs(plane * 3 + 10)
```

Many methods in `SpectralCube` allow you to extract subsets of relevant data, to make writing code like this easier:

- `SpectralCube.filled_data()`, `SpectralCube.unmasked_data()`, `SpectralCube.world()` all accept Numpy style slice syntax. For example, `cube.filled_data[0:3, :, :]` returns only the first 3 spectral channels of the cube, with masked elements replaced with `cube.fill_value`.

- `SpectralCube()` itself can be sliced to extract subcubes

- `SpectralCube.spectral_slab()` extracts a subset of spectral channels.

Many methods in `SpectralCube` iterate over smaller chunks of data, to avoid large memory allocations when working with big cubes. Some of these have a `how` keyword parameter, for fine-grained control over how much memory is accessed at once. `how='cube'` works with the entire array in memory, `how='slice'` works with one slice at a time, and `how='ray'` works with one ray at a time.

As a user, your best strategy for working with large datasets is to rely on built-in methods to `SpectralCube`, and to access data from `filled_data()` and `unmasked_data()` in smaller chunks if possible.

**Warning:** At the moment, `argmax()` and `argmin()`, are not optimized for handling large datasets.

### 2.2.2 Minimize Data Copying

Methods in `SpectralCube()` avoid copying as much as possible. For example, all of the following operations create new cubes or masks without copying any data:

```python
>>> mask = cube > 3
>>> slab = cube.spectral_slab(...)
>>> subcube = cube[0::2, 10:, 0:30]
>>> cube2 = cube.with_fill(np.nan)
>>> cube2 = cube.apply_mask(mask)
```

### 2.2.3 Minimize the number of passes over the data

Accessing memory-mapped arrays is much slower than a normal array, due to the overhead of reading from disk. Because of this, it is more efficient to perform computations that iterate over the data as few times as possible.

An even subtler issue pertains to how the 3D or 4D spectral cube is arranged as a 1D sequence of bytes in a file. Data access is much faster when it corresponds to a single contiguous scan of bytes on disk. For more information on this topic, see this tutorial on Numpy strides.

### 2.2.4 Recipe for large cube operations that can’t be done in memory

Sometimes, you will need to run full-cube operations that can’t be done in memory and can’t be handled by spectral-cube’s built in operations. An example might be converting your cube from Jy/beam to K when you have a very large (e.g., >10GB) cube.

Handling this sort of situation requires several manual steps. First, hard drive space needs to be allocated for the output data. Then, the cube must be manually looped over using a strategy that holds only limited data in memory.:
import shutil
from spectral_cube import SpectralCube
from astropy.io import fits

cube = SpectralCube.read('file.fits')

# this copy step is necessary to allocate memory for the output
shutil.copy('file.fits', 'newfile.fits')
outfh = fits.open('newfile.fits', mode='update')

jtok_factors = cube.jtok_factors()
for index, (slice, factor) in enumerate(zip(cube, factors)):
    outfh[index].data = slice * factor
outfh.flush()  # write the data to disk
outfh[0].header['BUNIT'] = 'K'
outfh.flush()  # write the header to disk

2.3 API Documentation

2.3.1 spectral_cube Package

This is an Astropy affiliated package.

Functions

```python
spectral_cube.test(package=None, test_path=None, args=None, plugins=None, verbose=False, pastebin=None, remote_data=False, pep8=False, pdb=False, coverage=False, open_files=False, **kwargs)
```

Run the tests using py.test. A proper set of arguments is constructed and passed to pytest.main.

**Parameters**

- `package` [str, optional] The name of a specific package to test, e.g. ‘io.fits’ or ‘utils’. If nothing is specified all default tests are run.
- `test_path` [str, optional] Specify location to test by path. May be a single file or directory. Must be specified absolutely or relative to the calling directory.
- `args` [str, optional] Additional arguments to be passed to pytest.main in the args keyword argument.
- `plugins` [list, optional] Plugins to be passed to pytest.main in the plugins keyword argument.
- `verbose` [bool, optional] Run tests in verbose mode.

---

2.3. API Documentation
Convenience option to turn on verbose output from `pytest`. Passing `True` is the same as specifying `--v` in `args`.

**pastesbin**

[['failed', 'all', None], optional] Convenience option for turning on `pytest` pastebin output. Set to `failed` to upload info for failed tests, or `all` to upload info for all tests.

**remote_data**

[bool, optional] Controls whether to run tests marked with `@remote_data`. These tests use online data and are not run by default. Set to `True` to run these tests.

**pep8**

[bool, optional] Turn on PEP8 checking via the `pytest-pep8` plugin and disable normal tests. Same as specifying `--pep8 -k pep8` in `args`.

**pdb**

[bool, optional] Turn on PDB post-mortem analysis for failing tests. Same as specifying `--pdb` in `args`.

**coverage**

[bool, optional] Generate a test coverage report. The result will be placed in the directory `htmlcov`.

**open_files**

[bool, optional] Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Requires the `psutil` package.

**parallel**

[int, optional] When provided, run the tests in parallel on the specified number of CPUs. If parallel is negative, it will use the all the cores on the machine. Requires the `pytest-xdist` plugin installed. Only available when using Astropy 0.3 or later.

**kwargs**

Any additional keywords passed into this function will be passed on to the astropy test runner. This allows use of test-related functionality implemented in later versions of astropy without explicitly updating the package template.

### Classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BooleanArrayMask</strong>(mask, wcs[, shape, include])</td>
<td>A mask defined as an array on a spectral cube WCS</td>
</tr>
<tr>
<td><strong>CompositeMask</strong>(mask1, mask2[, operation])</td>
<td>A combination of several masks.</td>
</tr>
<tr>
<td><strong>FunctionMask</strong>(function)</td>
<td>A mask defined by a function that is evaluated at run-time using the data passed to the mask.</td>
</tr>
<tr>
<td><strong>InvertedMask</strong>(mask)</td>
<td>A boolean mask defined by the evaluation of a comparison function between a fixed dataset and some other value.</td>
</tr>
<tr>
<td><strong>LazyComparisonMask</strong>(function, comparison_value)</td>
<td>A boolean mask defined by the evaluation of a function on a fixed dataset.</td>
</tr>
<tr>
<td><strong>LazyMask</strong>(function[, cube, data, wcs])</td>
<td>A boolean mask defined by the evaluation of a function on a fixed dataset.</td>
</tr>
</tbody>
</table>

Continued on next page
StokesSpectralCube(stokes_data[, mask, ...])  A class to store a spectral cube with multiple Stokes parameters.

VaryingResolutionSpectralCube(*args, **kwargs)  A variant of the SpectralCube class that has PSF (beam) information on a per-channel basis.

BooleanArrayMask

class spectral_cube.BooleanArrayMask(mask, wcs, shape=None, include=True)

Bases: spectral_cube.MaskBase

A mask defined as an array on a spectral cube WCS

Parameters

- mask: `numpy.ndarray`
  A boolean numpy ndarray

- wcs: `astropy.wcs.WCS`
  The WCS object

- shape: tuple
  The shape of the region the array is masking. This is required if `mask.ndim != data.ndim` to provide rules for how to broadcast the mask

Methods Summary

- **any()**
  Return a boolean array indicating which values should be excluded.

- **exclude([data, wcs, view])**
  Return a boolean array indicating which values should be excluded.
  If `view` is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.
  kwargs are passed to `_validate_wcs`

- **include([data, wcs, view])**
  Return a boolean array indicating which values should be included.

- **quicklook(view[, wcs, filename, use_aplpy, ...])**
  View a 2D slice of the mask, specified by `view`.

- **view([view])**
  Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.

- **with_spectral_unit(unit[, ...])**
  Get a BooleanArrayMask copy with a WCS in the modified unit

Methods Documentation

- **any()**

- **exclude(data=None, wcs=None, view=(), **kwargs)**
  Return a boolean array indicating which values should be excluded.
  If `view` is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.
  kwargs are passed to `_validate_wcs`

- **include(data=None, wcs=None, view=(), **kwargs)**
  Return a boolean array indicating which values should be included.
If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs

**quicklook**(view, wcs=None, filename=None, use_aplpy=True, aplpy_kwargs={})

View a 2D slice of the mask, specified by view.

**Parameters**

**view**
[tuple] Slicing to apply to the mask. Must return a 2D slice.

**wcs**
[astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.

**filename**

**use_aplpy**
[bool, optional] Try plotting with the aplpy package

**aplpy_kwargs**
[dict, optional] kwargs passed to FITSFigure.

**view**(view=())

Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.

Numpy’s convention is that masked=True means “masked out”

---

**Note:** I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.

**with_spectral_unit**(unit, velocity_convention=None, rest_value=None)

Get a BooleanArrayMask copy with a WCS in the modified unit

**Parameters**

**unit**
[u.Unit] Any valid spectral unit: velocity, (wave)length, or frequency. Only vacuum units are supported.

**velocity_convention**
[u.doppler_relativistic, u.doppler_radio, or u.doppler_optical] The velocity convention to use for the output velocity axis. Required if the output type is velocity.

**rest_value**
[u.Quantity] A rest wavelength or frequency with appropriate units. Required if output type is velocity. The cube’s WCS should include this already if the input type is velocity, but the WCS’s rest wavelength/frequency can be overridden with this parameter.

**CompositeMask**

class spectral_cube.CompositeMask(mask1, mask2, operation='and')

Bases: spectral_cube.MaskBase

A combination of several masks. The included masks are treated with the specified operation.
Parameters

mask1, mask2
[Masks] The two masks to composite

operation
[str] Either ‘and’ or ‘or’; the operation used to combine the masks

Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any()</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>exclude()</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>include()</td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td>quicklook()</td>
<td>View a 2D slice of the mask, specified by view.</td>
</tr>
<tr>
<td>view()</td>
<td>Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.</td>
</tr>
<tr>
<td>with_spectral_unit()</td>
<td>Get a CompositeMask copy in which each component has a WCS in the modified unit</td>
</tr>
</tbody>
</table>

Methods Documentation

any()

exclude(data=None, wcs=None, view=(), **kwargs)
Return a boolean array indicating which values should be excluded.

If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs

include(data=None, wcs=None, view=(), **kwargs)
Return a boolean array indicating which values should be included.

If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs

quicklook(view, wcs=None, filename=None, use_aplpy=True, aplpy_kwargs={})
View a 2D slice of the mask, specified by view.

Parameters

view
[tuple] Slicing to apply to the mask. Must return a 2D slice.

wcs
[astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.

filename

2.3. API Documentation
use_aplpy
[bool, optional] Try plotting with the aplpy package

aplpy_kwargs
[dict, optional] kwargs passed to FITSFigure.

view(view=())
Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.
Numpy’s convention is that masked=True means “masked out”

Note: I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.

with_spectral_unit(unit, velocity_convention=None, rest_value=None)
Get a CompositeMask copy in which each component has a WCS in the modified unit

Parameters

unit
[u.Unit] Any valid spectral unit: velocity, (wave)length, or frequency. Only vacuum units are supported.

velocity_convention
[u.doppler_relativistic, u.doppler_radio, or u.doppler_optical] The velocity convention to use for the output velocity axis. Required if the output type is velocity.

rest_value
[u.Quantity] A rest wavelength or frequency with appropriate units. Required if output type is velocity. The cube’s WCS should include this already if the input type is velocity, but the WCS’s rest wavelength/frequency can be overridden with this parameter.

FunctionMask

class spectral_cube.FunctionMask(function)
Bases: spectral_cube.MaskBase
A mask defined by a function that is evaluated at run-time using the data passed to the mask.
This function differs from LazyMask in the arguments which are passed to the function. Function Masks receive an array, wcs object, and view, whereas LazyMasks receive pre-sliced views into an array specified at mask-creation time.

Parameters

function
[callable] The function to evaluate the mask. The call signature should be function(data, wcs, slice) where data and wcs are the arguments that get passed to e.g. include, exclude, _filled, and _flattened. The function should return a boolean array, where True values indicate that which pixels are valid / unaffected by masking.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any()</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>exclude([data, wcs, view])</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>include([data, wcs, view])</td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td>quicklook(view[, wcs, filename, use_aplpy, ...])</td>
<td>View a 2D slice of the mask, specified by view.</td>
</tr>
<tr>
<td>view([view])</td>
<td>Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.</td>
</tr>
<tr>
<td>with_spectral_unit(unit[, ...])</td>
<td>Functional masks do not have WCS defined, so this simply returns a copy of the current mask in order to be consistent with with_spectral_unit from other Masks</td>
</tr>
</tbody>
</table>

### Methods Documentation

**any()**

```python
def any():
    return True
```

**exclude**(data=None, wcs=None, view=(), **kwargs)

Return a boolean array indicating which values should be excluded.

If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs

**include**(data=None, wcs=None, view=(), **kwargs)

Return a boolean array indicating which values should be included.

If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs

**quicklook**(view[, wcs, filename, use_aplpy, ...])

View a 2D slice of the mask, specified by view.

**view**(view=())

Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.

**Parameters**

- **view**
  - [tuple] Slicing to apply to the mask. Must return a 2D slice.

- **wcs**
  - [astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.

- **filename**

- **use_aplpy**
  - [bool, optional] Try plotting with the aplpy package

- **aplpy_kwargs**
  - [dict, optional] kwargs passed to FITSFigure.
Numpy’s convention is that masked=True means “masked out”

**Note**: I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.

```python
with_spectral_unit(unit, velocity_convention=None, rest_value=None)
```

Functional masks do not have WCS defined, so this simply returns a copy of the current mask in order to be consistent with `with_spectral_unit` from other Masks

### InvertedMask

**class** spectral_cube.InvertedMask(mask)

Bases: spectral_cube.MaskBase

**Methods Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>any()</strong></td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td><strong>exclude([data, wcs, view])</strong></td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td><strong>include([data, wcs, view])</strong></td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td><strong>quicklook(view[, wcs, filename, use_aplpy, ...])</strong></td>
<td>View a 2D slice of the mask, specified by view.</td>
</tr>
<tr>
<td><strong>view([view])</strong></td>
<td>Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.</td>
</tr>
<tr>
<td><strong>with_spectral_unit(unit[,...])</strong></td>
<td>Get an InvertedMask copy with a WCS in the modified unit</td>
</tr>
</tbody>
</table>

**Methods Documentation**

**any()**

```python
exclude(data=None, wcs=None, view=(), **kwargs)
```

Return a boolean array indicating which values should be excluded.

If `view` is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

`kwargs` are passed to `_validate_wcs`

```python
include(data=None, wcs=None, view=(), **kwargs)
```

Return a boolean array indicating which values should be included.

If `view` is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

`kwargs` are passed to `_validate_wcs`

```python
quicklook(view[, wcs, filename, use_aplpy,...])
```

View a 2D slice of the mask, specified by view.

**Parameters**
view
[tuple] Slicing to apply to the mask. Must return a 2D slice.

wcs
[astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.

filename

use_aplpy
[bool, optional] Try plotting with the aplpy package

aplpy_kwargs
[dict, optional] kwargs passed to FITSFigure.

view(view=())
Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.

Numpy’s convention is that masked=True means “masked out”

Note: I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.

with_spectral_unit(unit, velocity_convention=None, rest_value=None)
Get an InvertedMask copy with a WCS in the modified unit

Parameters

unit
[u.Unit] Any valid spectral unit: velocity, (wave)length, or frequency. Only vacuum units are supported.

velocity_convention
[u.doppler_relativistic, u.doppler_radio, or u.doppler_optical] The velocity convention to use for the output velocity axis. Required if the output type is velocity.

rest_value
[u.Quantity] A rest wavelength or frequency with appropriate units. Required if output type is velocity. The cube’s WCS should include this already if the input type is velocity, but the WCS’s rest wavelength/frequency can be overridden with this parameter.

LazyComparisonMask

class spectral_cube.LazyComparisonMask(function, comparison_value, cube=None, data=None, wcs=None)
Bases: spectral_cube.LazyMask
A boolean mask defined by the evaluation of a comparison function between a fixed dataset and some other value.
This is conceptually similar to the LazyMask but it will ensure that the comparison value can be compared to the data

Parameters

function
[callable] The function to apply to data. This method should accept a numpy array.
which will be the data array passed to `__init__`, and a second argument also passed to `__init__`. It should return a boolean array, where `True` values indicate that which pixels are valid/unaffected by masking.

**comparison_value**

[Float or array] The comparison value for the array

**data**

[array-like] The array to evaluate function on. This should support Numpy-like slicing syntax.

**wcs**

[WCS] The WCS of the input data, which is used to define the coordinates for which the boolean mask is defined.

### Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any()</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>exclude()</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>include()</td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td>quicklook()</td>
<td>View a 2D slice of the mask, specified by view.</td>
</tr>
<tr>
<td>view()</td>
<td>Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.</td>
</tr>
<tr>
<td>with_spectral_unit()</td>
<td>Get a LazyComparisonMask copy with a WCS in the modified unit</td>
</tr>
</tbody>
</table>

### Methods Documentation

**any()**

**exclude**(data=None, wcs=None, view=(), **kwargs)

Return a boolean array indicating which values should be excluded.

If `view` is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

`kwargs` are passed to `_validate_wcs`

**include**(data=None, wcs=None, view=(), **kwargs)

Return a boolean array indicating which values should be included.

If `view` is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

`kwargs` are passed to `_validate_wcs`

**quicklook**(view, wcs=None, filename=None, use_aplpy=True, aplpy_kwargs={})

View a 2D slice of the mask, specified by view.

**Parameters**

**view**

[tuple] Slicing to apply to the mask. Must return a 2D slice.
wcs
[astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.

filename

use_aplpy
[bool, optional] Try plotting with the aplpy package

aplpy_kwargs
[dict, optional] kwargs passed to FITSFigure.

view(view=())
Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.

Numpy’s convention is that masked=True means “masked out”

Note: I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.

with_spectral_unit(unit, velocity_convention=None, rest_value=None)
Get a LazyComparisonMask copy with a WCS in the modified unit

LazyMask
class spectral_cube.LazyMask(function, cube=None, data=None, wcs=None)
Bases: spectral_cube.MaskBase
A boolean mask defined by the evaluation of a function on a fixed dataset.

This is conceptually identical to a fixed boolean mask as in BooleanArrayMask but defers the evaluation of the mask until it is needed.

Parameters

function
[callable] The function to apply to data. This method should accept a numpy array, which will be a subset of the data array passed to __init__. It should return a boolean array, where True values indicate that which pixels are valid/unaffected by masking.

data
[array-like] The array to evaluate function on. This should support Numpy-like slicing syntax.

wcs
[WCS] The WCS of the input data, which is used to define the coordinates for which the boolean mask is defined.

Methods Summary

any()
exclude([data, wcs, view]) Return a boolean array indicating which values should be excluded.

Continued on next page
Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include([data, wcs, view])</code></td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td><code>quicklook(view[, wcs, filename, use_aplpy, ...])</code></td>
<td>View a 2D slice of the mask, specified by view.</td>
</tr>
<tr>
<td><code>view([view])</code></td>
<td>Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.</td>
</tr>
<tr>
<td><code>with_spectral_unit(unit[, ...])</code></td>
<td>Get a LazyMask copy with a WCS in the modified unit</td>
</tr>
</tbody>
</table>

Methods Documentation

any()

exclude(data=None, wcs=None, view=(), **kwargs)
   Return a boolean array indicating which values should be excluded.
   If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.
   kwars are passed to _validate_wcs

include(data=None, wcs=None, view=(), **kwargs)
   Return a boolean array indicating which values should be included.
   If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.
   kwars are passed to _validate_wcs

quicklook(view, wcs=None, filename=None, use_aplpy=True, aplpy_kwargs={})
   View a 2D slice of the mask, specified by view.
   Parameters
   view
      [tuple] Slicing to apply to the mask. Must return a 2D slice.
   wcs
      [astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.
   filename
   use_aplpy
      [bool, optional] Try plotting with the aplpy package
   aplpy_kwargs
      [dict, optional] kwargs passed to FITSFigure.
   view()
      Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.
      Numpy’s convention is that masked=True means “masked out”

Note: I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.
with_spectral_unit(unit, velocity_convention=None, rest_value=None)

Get a LazyMask copy with a WCS in the modified unit

Parameters

unit
[u.Unit] Any valid spectral unit: velocity, (wave)length, or frequency. Only vacuum units are supported.

velocity_convention
[u.doppler_relativistic, u.doppler_radio, or u.doppler_optical] The velocity convention to use for the output velocity axis. Required if the output type is velocity.

rest_value
[u.Quantity] A rest wavelength or frequency with appropriate units. Required if output type is velocity. The cube’s WCS should include this already if the input type is velocity, but the WCS’s rest wavelength/frequency can be overridden with this parameter.

MaskBase

class spectral_cube.MaskBase

Bases: object

Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any()</td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td>exclude()</td>
<td>Return a boolean array indicating which values should be excluded.</td>
</tr>
<tr>
<td>include()</td>
<td>Return a boolean array indicating which values should be included.</td>
</tr>
<tr>
<td>quicklook()</td>
<td>View a 2D slice of the mask, specified by view.</td>
</tr>
<tr>
<td>view()</td>
<td>Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array.</td>
</tr>
</tbody>
</table>

Methods Documentation

any()

exclude(data=None, wcs=None, view=(), **kwargs)

Return a boolean array indicating which values should be excluded.

If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs

include(data=None, wcs=None, view=(), **kwargs)

Return a boolean array indicating which values should be included.

If view is passed, only the sliced mask will be returned, which avoids having to load the whole mask in memory. Otherwise, the whole mask is returned in-memory.

kwargs are passed to _validate_wcs
quicklook(view, wcs=None, filename=None, use_aplpy=True, aplpy_kwargs={})

View a 2D slice of the mask, specified by view.

**Parameters**

- **view**
  - [tuple] Slicing to apply to the mask. Must return a 2D slice.

- **wcs**
  - [astropy.wcs.WCS, optional] WCS object to use in plotting the mask slice.

- **filename**

- **use_aplpy**
  - [bool, optional] Try plotting with the aplpy package

- **aplpy_kwargs**
  - [dict, optional] kwargs passed to FITSFigure.

**view(view=())**

Compatibility tool: if a numpy.ma.ufunc is run on the mask, it will try to grab a view of the mask, which needs to appear to numpy as a true array. This can be important for, e.g., plotting.

Numpy’s convention is that masked=True means “masked out”

---

**Note:** I don’t know if there are broader concerns or consequences from including this ‘view’ tool here.

---

**OneDSpectrum**

```python
class spectral_cube.OneDSpectrum
```

**Attributes Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T</code></td>
<td>Same as self.transpose(), except that self is returned if self.ndim &lt; 2.</td>
</tr>
<tr>
<td><code>base</code></td>
<td>Base object if memory is from some other object.</td>
</tr>
<tr>
<td><code>ctypes</code></td>
<td>An object to simplify the interaction of the array with the ctypes module.</td>
</tr>
<tr>
<td><code>data</code></td>
<td>Python buffer object pointing to the start of the array’s data.</td>
</tr>
<tr>
<td><code>dtype</code></td>
<td>Data-type of the array’s elements.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Information about the memory layout of the array.</td>
</tr>
<tr>
<td><code>imag</code></td>
<td>The imaginary part of the array.</td>
</tr>
<tr>
<td><code>info</code></td>
<td>Container for meta information like name, description, format.</td>
</tr>
<tr>
<td><code>itemsize</code></td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td><code>nbytes</code></td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td><code>ndim</code></td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td><code>real</code></td>
<td>The real part of the array.</td>
</tr>
</tbody>
</table>

---

Continued on next page
Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>shape</th>
<th>Tuple of array dimensions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</td>
</tr>
</tbody>
</table>

**Methods Summary**

- **all([axis, out, keepdims])**
  Returns True if all elements evaluate to True.

- **any([axis, out, keepdims])**
  Returns True if any of the elements of a evaluate to True.

- **argmax([axis, out])**
  Return indices of the maximum values along the given axis.

- **argmin([axis, out])**
  Return indices of the minimum values along the given axis.

- **argpartition(kth[, axis, kind, order])**
  Returns the indices that would partition this array.

- **argsort([axis, kind, order])**
  Return indices that would sort this array.

- **astype(dtype[, order, casting, subok, copy])**
  Copy of the array, cast to a specified type.

- **byteswap([inplace])**
  Swap the bytes of the array elements

- **choose(choices[, out, mode])**
  Use an index array to construct a new array from a set of choices.

- **clip([min, max, out])**
  Return an array whose values are limited to [min, max].

- **compress(condition[, axis, out])**
  Return selected slices of this array along given axis.

- **conj()**
  Complex-conjugate all elements.

- **conjugate()**
  Return the complex conjugate, element-wise.

- **copy([order])**
  Return a copy of the array.

- **cumprod([axis, dtype, out])**
  Return the cumulative product of the elements along the given axis.

- **cumsum([axis, dtype, out])**
  Return the cumulative sum of the elements along the given axis.

- **decompose([bases])**
  Generates a new Quantity with the units decomposed.

- **diagonal([offset, axis1, axis2])**
  Return specified diagonals.

- **diff(n, axis)**
  Dot product of two arrays.

- **dump(file)**
  Dump a pickle of the array to the specified file.

- **dumps()**
  Returns the pickle of the array as a string.

- **ediff1d([to_end, to_begin])**
  Fill the array with a scalar value.

- **filled([fill_value])**
  Return a copy of the array collapsed into one dimension.

- **from_hdu(hdu)**
  Return a OneDSpectrum from a FITS HDU or HDU list.

- **getfield(dtype[, offset])**
  Returns a field of the given array as a certain type.

- **insert(obj, values[, axis])**
  Insert values along the given axis before the given indices and return a new Quantity object.

- **item(*args)**
  Copy an element of an array to a standard Python scalar and return it.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>itemset(*args)</code></td>
<td>Insert scalar into an array (scalar is cast to array’s dtype, if possible)</td>
</tr>
<tr>
<td><code>max([axis, out, keepdims])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>mean([axis, dtype, out, keepdims])</code></td>
<td>Returns the average of the array elements along given axis.</td>
</tr>
<tr>
<td><code>min([axis, out, keepdims])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>nansum([axis, out, keepdims])</code></td>
<td>Return the sum of the array elements along given axis.</td>
</tr>
<tr>
<td><code>newbyteorder([new_order])</code></td>
<td>Return the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>partition(kth[, axis, kind, order])</code></td>
<td>Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.</td>
</tr>
<tr>
<td><code>prod([axis, dtype, out, keepdims])</code></td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>ptp([axis, out, keepdims])</code></td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
<tr>
<td><code>put(indices, values[, mode])</code></td>
<td>Set <code>a.flat[n] = values[n]</code> for all <code>n in indices</code>.</td>
</tr>
<tr>
<td><code>quicklook([filename, drawstyle])</code></td>
<td>Plot the spectrum with current spectral units in the currently open figure.</td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>reshape(shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>resize(new_shape[, refcheck])</code></td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td><code>round([decimals, out])</code></td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td><code>searchsorted(v[, side, sorter])</code></td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td><code>setfield(val, dtype[, offset])</code></td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td><code>setflags([write, align, uic])</code></td>
<td>Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.</td>
</tr>
<tr>
<td><code>shrink_mask()</code></td>
<td>Copy of the numpy masked_array shrink_mask method.</td>
</tr>
<tr>
<td><code>sort([axis, kind, order])</code></td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td><code>spectral_interpolate(spectral_grid[, ...])</code></td>
<td>Resample the spectrum onto a specific grid</td>
</tr>
<tr>
<td><code>spectral_smooth(kernel[, convolve])</code></td>
<td>Smooth the spectrum</td>
</tr>
<tr>
<td><code>squeeze([axis])</code></td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td><code>std([axis, dtype, out, ddof, keepdims])</code></td>
<td>Returns the standard deviation of the array elements along given axis.</td>
</tr>
<tr>
<td><code>sum([axis, dtype, out, keepdims])</code></td>
<td>Return the sum of the array elements along the given axis.</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td><code>take(indices[, axis, out, mode])</code></td>
<td>Return an array formed from the elements of a at the given indices.</td>
</tr>
<tr>
<td><code>to(unit[, equivalencies])</code></td>
<td>Return a new <code>OneDSpectrum</code> of the same class with the specified unit.</td>
</tr>
</tbody>
</table>
Table 11 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_string([unit, precision, format, subfmt])</code></td>
<td>Generate a string representation of the quantity and its unit.</td>
</tr>
<tr>
<td><code>to_value([unit, equivalencies])</code></td>
<td>The numerical value, possibly in a different unit.</td>
</tr>
<tr>
<td><code>tobytes([order])</code></td>
<td>Construct Python bytes containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>tofile(fid[, sep, format])</code></td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td><code>tostring([order])</code></td>
<td>Construct Python bytes containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>trace(offset, axis1, axis2, dtype, out)[]</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>var([axis, dtype, out, ddof, keepdims])</code></td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td><code>view([dtype, type])</code></td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td><code>with_beam(beam)</code></td>
<td>Attach a new beam object to the OneDSpectrum.</td>
</tr>
<tr>
<td><code>with_fill_value(fill_value)</code></td>
<td>Create a new <code>OneDSpectrum</code> with a different <code>fill_value</code>.</td>
</tr>
<tr>
<td><code>with_spectral_unit(unit[, ...])</code></td>
<td></td>
</tr>
<tr>
<td><code>write(filename[, format, overwrite])</code></td>
<td>Write the lower dimensional object to a file.</td>
</tr>
</tbody>
</table>

Attributes Documentation

**T**

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

**Examples**

```python
>>> 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.])
```

**base**

Base object if memory is from some other object.

**Examples**

The base of an array that owns its memory is `None`:

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```python
>>> x = np.array([1, 2, 3, 4])
>>> x
array([1, 2, 3, 4])
>>> x[1] = 10
>>> x
array([ 1, 10, 3, 4])
```

2.3. API Documentation
```python
>>> 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):

_ctypes.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].

Note that unlike data_as, a reference will not be kept to the array: code like ctypes.c_void_p((a + b).ctypes.data) will result in a pointer to a deallocated array, and should be spelt (a + b).ctypes.data_as(ctypes.c_void_p)

_ctypes.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 ctypes.c_int, ctypes.c_long, or ctypes.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.

_ctypes.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.

_ctypes.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))`.

The returned pointer will keep a reference to the array.

```python
_ctypes.shape_as(obj)
```

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

```python
_ctypes.strides_as(obj)
```

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

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

```python
>>> 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 dtypes object]
```

### See also:
Examples

```python
>>> 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.

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 WRITEBACKIFCOPY, 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`.
- WRITEBACKIFCOPY 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 writable 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.

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 writable array may be subsequently locked while the base array remains writable. (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.

**WRITEBACKIFCOPY (X)**
This array is a copy of some other array. The C-API function `PyArray.ResolveWritebackIfCopy` must be called before deallocating to the base array will be updated with the contents of this array.

**UPDATEIFCOPY (U)**
(Deprecated, use WRITEBACKIFCOPY) 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.

imag
The imaginary part of the array.

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

info
Container for meta information like name, description, format. This is required when the object is used as a mixin column within a table, but can be used as a general way to store meta information.

**itemsize**
Length of one array element in bytes.

**Examples**

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
```
>>> 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**

```python
>>> 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**

```python
>>> 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.

**See also:**

- `numpy.real`
  
equivalent function

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

**shape**

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with `numpy.reshape`, one of the new
shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

**See also:**

*numpy.reshape*
  similar function

*ndarray.reshape*
  similar method

**Examples**

```python
g.x = np.array([1, 2, 3, 4])
g.x.shape
(4,)
g.y = np.zeros((2, 3, 4))
g.y.shape
(2, 3, 4)
g.y.shape = (3, 8)
g.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.]])
g.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.

Equal to np.prod(a.shape), i.e., the product of the array’s dimensions.

**Notes**

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested np.prod(a.shape), which returns an instance of np.int_), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

**Examples**

```python
g.x = np.zeros((3, 5, 2), dtype=np.complex128)
g.x.size
30
```

**strides**

Tuple of bytes to step in each dimension when traversing an array.
The byte offset of element \((i[0], i[1], \ldots, i[n])\) in an array \(a\) is:

\[
\text{offset} = \text{sum}(\text{np.array}(i) \times a.\text{strides})
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

See also:

numpy.lib.stride_tricks.as_strided

Notes

Imagine an array of 32-bit integers (each 4 bytes):

\[
x = \text{np.array}([[0, 1, 2, 3, 4],
    [5, 6, 7, 8, 9]], \text{dtype=\text{np.int32}})
\]

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 \((20, 4)\).

Examples

```python
>>> 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
```

Methods Documentation

`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`(*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`(*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`(*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`(*kth, axis=-1, kind='introselect', order=None*)

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

**See also:**

- `numpy.argpartition` equivalent function

`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(dtype, order='K', casting='unsafe', subok=True, copy=True)

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[small 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.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use a.real. astype(t).

**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.
Examples

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

byteswap(inplace=False)

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

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']

>>> A.byteswap(inplace=True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

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(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(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()
   Complex-conjugate all elements.
   Refer to numpy.conjugate for full documentation.
   See also:

   numpy.conjugate
      equivalent function
```

```
conjugate()
   Return the complex conjugate, element-wise.
   Refer to numpy.conjugate for full documentation.
   See also:

   numpy.conjugate
      equivalent function
```

```
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 numpy.copy() are very similar, but have different default values for their order= arguments.)

   See also:

   numpy.copy, numpy.copyto
```

```
Examples
```

```python
>>> x = np.array([[1,2,3],[4,5,6]], order='F')

>>> y = x.copy()
```

```
```python
>>> 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(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(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
decompose(bases=[])  
Generates a new `Quantity` with the units decomposed. Decomposed units have only irreducible units in them (see `astropy.units.UnitBase.decompose`).

Parameters

- bases  
  [sequence of `UnitBase`, optional] The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it’s not possible to do so.

Returns

- newq  
  `[Quantity]` A new object equal to this quantity with units decomposed.
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 `numpy.diagonal()` for full documentation.

See also:
**numpy.diagonal**

   equivalent function

**diff**(*n=1, axis=-1*)

**dot**(*b, out=None*)

   Dot product of two arrays.

   Refer to *numpy.dot* for full documentation.

   **See also:**

   *numpy.dot*

   equivalent function

**Examples**

```python
>>> 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:

```python
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

**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**()

   Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

   **Parameters**

   * None

**ediff1d**(*to_end=None, to_begin=None*)

**fill**(*value*)

   Fill the array with a scalar value.

   **Parameters**

   * value
     [scalar] All elements of a will be assigned this value.
Examples

```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
```

```python
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])
```

`filled(fill_value=None)`

`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

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
```

```python
>>> a = np.array([[1, 2, 3, 4]])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

`static from_hdu(hdu)`

Return a OneDSpectrum from a FITS HDU or HDU list.

`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**

[Detection of 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

```python
>>> 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:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```

**insert**(obj, values, axis=None)

Insert values along the given axis before the given indices and return a new `Quantity` object.

This is a thin wrapper around the `numpy.insert` function.

Parameters

**obj**

[int, slice or sequence of ints] Object that defines the index or indices before which values is inserted.

**values**

[array-like] Values to insert. If the type of values is different from that of quantity, values is converted to the matching type. values should be shaped so that it can be broadcast appropriately The unit of values must be consistent with this quantity.

**axis**

[int, optional] Axis along which to insert values. If axis is None then the quantity array is flattened before insertion.

Returns

**out**

[Quantity] A copy of quantity with values inserted. Note that the insertion does not occur in-place: a new quantity array is returned.
Examples

```python
>>> import astropy.units as u

>>> q = [1, 2] * u.m
>>> q.insert(0, 50 * u.cm)
<Quantity [0.5, 1., 2.] m>

>>> q = [[1, 2], [3, 4]] * u.m
>>> q.insert(1, [10, 20] * u.m, axis=0)
<Quantity [[ 1., 2.],
          [10., 20.],
          [ 3., 4.]] m>

>>> q.insert(1, 10 * u.m, axis=1)
<Quantity [[ 1., 10., 2.],
          [ 3., 10., 4.]] m>
```

`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))
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

\textbf{itemset}(\*\texttt{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 \texttt{item}. Then, \texttt{a.itemset(*args)} is equivalent to but faster than \texttt{a[\texttt{args}] = item}. The item should be a scalar value and \texttt{args} must select a single item in the array \texttt{a}.

**Parameters**

*\texttt{args}*

[Arguments] If one argument: a scalar, only used in case \texttt{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, \texttt{itemset} provides some speed increase for placing a scalar into a particular location in an \texttt{ndarray}, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using \texttt{itemset} \texttt{(and \texttt{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**

max(\texttt{axis=None, out=None, keepdims=False})

Return the maximum along a given axis.

Refer to \texttt{numpy.amax} for full documentation.

See also:
**numpy.amax**
equivalent function

**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(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

**nansum(axis=None, out=None, keepdims=False)**

**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’, ‘T’ - 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()
Return the indices of the elements that are non-zero.
Refer to numpy.nonzero for full documentation.
See also:
numpy.nonzero equivalent function

partition(kth, axis=-1, kind='introselect', order=None)
Rearranges the elements in the array 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.
New in version 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 of 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 to 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

```python
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
array([2, 1, 3, 4])

array([1, 2, 3, 4])
```

```
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(axis=None, out=None, keepdims=False)
Peak to peak (maximum - minimum) value along a given axis.
Refer to `numpy.ptp` for full documentation.

See also:
`numpy.ptp` equivalent function
```

```
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
```

```
quicklook(filename=None, drawstyle='steps-mid', **kwargs)
Plot the spectrum with current spectral units in the currently open figure
kwargs are passed to `matplotlib.pyplot.plot`

Parameters

filename
[None or Non] Optional - the filename to save the quicklook to.
```

```
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(repeats, axis=None)
    Repeat elements of an array.
    Refer to numpy.repeat for full documentation.
    See also:
        numpy.repeat
            equivalent function

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

Notes

Unlike the free function numpy.reshape, this method on ndarray allows the elements of the shape parameter to be passed in as separate arguments. For example, a.reshape(10, 11) is equivalent to a.reshape((10, 11)).

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. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.
    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:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```python
>>> 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:

```python
>>> 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...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ...
ValueError: cannot resize an array that has been referenced ...
```

Unless refcheck is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```
round(\textit{decimals=0, out=None})

Return a with each element rounded to the given number of decimals.

Refer to \texttt{numpy.around} for full documentation.

See also:

\texttt{numpy.around}

equivalent function

searchsorted(\textit{v, side='left', sorter=None})

Find indices where elements of \textit{v} should be inserted in \textit{a} to maintain order.

For full documentation, see \texttt{numpy.searchsorted}

See also:

\texttt{numpy.searchsorted}

equivalent function

setfield(\textit{val, dtype, offset=0})

Put a value into a specified place in a field defined by a data-type.

Place \textit{val} into \textit{a}'s field defined by \textit{dtype} and beginning \textit{offset} bytes into the field.

Parameters

\texttt{val}  
[object] Value to be placed in field.

\texttt{dtype}  
[dtype object] Data-type of the field in which to place \textit{val}.

\texttt{offset}  
[int, optional] The number of bytes into the field at which to place \textit{val}.

Returns

None

See also:

getField

Examples

```python
>>> 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]])
```
setflags\((write=None, align=None, uic=None)\)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY 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 WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags 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 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, 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) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIFCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```python
>>> y
array([[3, 1, 7],
       [2, 0, 0]])
```
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False

>>> y.setflags(write=0, align=0)

>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False

>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True

\[8, 5, 9]\]

shrink_mask()
Copy of the numpy masked_array shrink_mask method. This is essentially a hack needed for matplotlib to show images.

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', 'stable'}, 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

```python
>>> 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:

```python
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', '<f4'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '<f4'), ('y', '<i4')])
```

**spectral_interpolate**(*spectral_grid*, *suppress_smooth_warning*=`False`, *fill_value*=`None`)
Resample the spectrum onto a specific grid

**Parameters**

**spectral_grid**
[array] An array of the spectral positions to regrid onto

**suppress_smooth_warning**
[bool] If disabled, a warning will be raised when interpolating onto a grid that does not nyquist sample the existing grid. Disable this if you have already appropriately smoothed the data.

**fill_value**
[float] Value for extrapolated spectral values that lie outside of the spectral range defined in the original data. The default is to use the nearest spectral channel in the cube.

**Returns**

**spectrum**
[OneDSpectrum]

**spectral_smooth**(*kernel*, *convolve*=`<function convolve>`, **kwargs)
Smooth the spectrum

**Parameters**
kernel
[Kernel1D] A 1D kernel from astropy

convolve
[function] The astropy convolution function to use, either astropy.convolution.convolve or astropy.convolution.convolve_fft

kwargs
[dict] Passed to the convolve function

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(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(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(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(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
to(unit, equivalencies=[],)
Return a new OneDSpectrum of the same class with the specified unit. See astropy.units.Quantity.to for further details.

to_string(unit=None, precision=None, format=None, subfmt=None)
Generate a string representation of the quantity and its unit.

The behavior of this function can be altered via the numpy.set_printoptions function and its various keywords. The exception to this is the threshold keyword, which is controlled via the [units.quantity] configuration item latex_array_threshold. This is treated separately because the numpy default of 1000 is too big for most browsers to handle.

Parameters

unit
[UnitBase, optional] Specifies the unit. If not provided, the unit used to initialize the quantity will be used.

precision
[numERIC, optional] The level of decimal precision. If None, or not provided, it will be determined from NumPy print options.

format
[str, optional] The format of the result. If not provided, an unadorned string is returned. Supported values are:
• ‘latex’: Return a LaTeX-formatted string

subfmt
[str, optional] Subformat of the result. For the moment, only used for format=”latex”. Supported values are:
• ‘inline’: Use $ ... $ as delimiters.
• ‘display’: Use $\displaystyle ... $ as delimiters.

Returns

lstr
A string with the contents of this Quantity

to_value(unit=None, equivalencies=[])
The numerical value, possibly in a different unit.

Parameters

unit
[UnitBase instance or str, optional] The unit in which the value should be given. If not given or None, use the current unit.

equivalencies
[list of equivalence pairs, optional] A list of equivalence pairs to try if the units are not directly convertible (see Equivalencies). If not provided or [], class default equivalencies will be used (none for Quantity, but may be set for subclasses). If None, no equivalencies will be applied at all, not even any set globally or within a context.

Returns
value

[ndarray or scalar] The value in the units specified. For arrays, this will be a view of the
data if no unit conversion was necessary.

See also:

to

Get a new instance in a different unit.

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.

New in version 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

```python
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x01\x00\x00\x02\x00\x00\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
'b'\x00\x01\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

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.

When fid is a file object, array contents are directly written to the file, bypassing the file object’s write method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support fileno() (e.g., BytesIO).

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, \(a = \text{np.array}(a.\text{tolist}())\).

Examples

```python
>>> a = np.array([[1, 2]]
>>> a.tolist()
[1, 2]
```

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()

[['1, 2'], ['3, 4']]
```

 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

```python
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x00
>>> x.tobytes(‘C’) == x.tobytes()
True
>>> x.tobytes(‘F’)
'b'\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00
```

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(*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 “conven-
  nience” 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**

```python
c>>> a = np.array([[1, 2], [3, 4]])
c>>> a
array([[1, 2],
       [3, 4]])
c>>> a.transpose()
array([[1, 3],
       [2, 4]])
c>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
c>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

**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** *(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

```python
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```python
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
```

Creating a view on a structured array so it can be used in calculations

```python
>>> 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

```python
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```python
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```python
>>> 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.:
with_beam(beam)

Attach a new beam object to the OneDSpectrum.

Parameters

beam

with_fill_value(fill_value)

Create a new OneDSpectrum with a different fill_value.

with_spectral_unit(unit, velocity_convention=None, rest_value=None)

write(filename, format=None, overwrite=False)

Write the lower dimensional object to a file.

Parameters

filename
[str] The path to write the file to

format
[str] The kind of file to write. (Currently limited to ‘fits’)

overwrite
[bool] If True, overwrite filename if it exists

Projection

class spectral_cube.Projection

Bases: spectral_cube.lower_dimensional_structures.LowerDimensionalObject, spectral_cube.base_class.SpatialCoordMixinClass, spectral_cube.base_class.MaskableArrayMixinClass, spectral_cube.base_class.BeamMixinClass

Attributes Summary

T
Same as self.transpose(), except that self is returned if self.ndim < 2.

base
Base object if memory is from some other object.

ctypes
An object to simplify the interaction of the array with the ctypes module.

data
Python buffer object pointing to the start of the array’s data.

dtype
Data-type of the array’s elements.
Table 12 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>Information about the memory layout of the array.</td>
</tr>
<tr>
<td>imag</td>
<td>The imaginary part of the array.</td>
</tr>
<tr>
<td>info</td>
<td>Container for meta information like name, description, format.</td>
</tr>
<tr>
<td>itemsize</td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td>nbytes</td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td>ndim</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>real</td>
<td>The real part of the array.</td>
</tr>
<tr>
<td>shape</td>
<td>Tuple of array dimensions.</td>
</tr>
<tr>
<td>size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</td>
</tr>
</tbody>
</table>

Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all(axis, out, keepdims)</td>
<td>Returns True if all elements evaluate to True.</td>
</tr>
<tr>
<td>any(axis, out, keepdims)</td>
<td>Returns True if any of the elements of a evaluate to True.</td>
</tr>
<tr>
<td>argmax(axis, out)</td>
<td>Return indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td>argmin(axis, out)</td>
<td>Return indices of the minimum values along the given axis of a.</td>
</tr>
<tr>
<td>argpartition(kth[, axis, kind, order])</td>
<td>Returns the indices that would partition this array.</td>
</tr>
<tr>
<td>argsort([axis, kind, order])</td>
<td>Returns the indices that would sort this array.</td>
</tr>
<tr>
<td>astype(dtype[, order, casting, subok, copy])</td>
<td>Copy of the array, cast to a specified type.</td>
</tr>
<tr>
<td>byteswap([inplace])</td>
<td>Swap the bytes of the array elements.</td>
</tr>
<tr>
<td>choose(choices[, out, mode])</td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td>clip([min, max, out])</td>
<td>Return an array whose values are limited to [min, max].</td>
</tr>
<tr>
<td>compress(condition[, axis, out])</td>
<td>Return selected slices of this array along given axis.</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex-conjugate all elements.</td>
</tr>
<tr>
<td>conjugate()</td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td>convolve_to(beam[, convolve])</td>
<td>Convolve the image to a specified beam.</td>
</tr>
<tr>
<td>copy([order])</td>
<td>Return a copy of the array.</td>
</tr>
<tr>
<td>cumprod(axis, dtype, out)</td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td>cumsum(axis, dtype, out)</td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td>decompose([bases])</td>
<td>Generates a new Quantity with the units decomposed.</td>
</tr>
<tr>
<td>diagonal(offset, axis1, axis2)</td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td>diff([n, axis])</td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td>dump(file)</td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td>dumps()</td>
<td>Returns the pickle of the array as a string.</td>
</tr>
<tr>
<td>ediff1d([to_end, to_begin])</td>
<td>Fill the array with a scalar value.</td>
</tr>
<tr>
<td>filled([fill_value])</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 13 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flatten([order])</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>from_hdu(hdu)</code></td>
<td>Return a projection from a FITS HDU.</td>
</tr>
<tr>
<td><code>getfield(dtype[, offset])</code></td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td><code>insert(obj, values[, axis])</code></td>
<td>Insert values along the given axis before the given indices and return a new <code>Quantity</code> object.</td>
</tr>
<tr>
<td><code>item(*args)</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>itemset(*args)</code></td>
<td>Insert scalar into an array (scalar is cast to array's dtype, if possible)</td>
</tr>
<tr>
<td><code>max([axis, out, keepdims])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>mean([axis, dtype, out, keepdims])</code></td>
<td>Returns the average of the array elements along given axis.</td>
</tr>
<tr>
<td><code>min([axis, out, keepdims])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>nansum([axis, out, keepdims])</code></td>
<td>Returns the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td><code>newbyteorder([new_order])</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.</td>
</tr>
<tr>
<td><code>prod([axis, dtype, out, keepdims])</code></td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>ptp([axis, out, keepdims])</code></td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
<tr>
<td><code>put(indices, values[, mode])</code></td>
<td>Set a.flat[n] = values[n] for all n in indices.</td>
</tr>
<tr>
<td><code>quicklook([filename, use_aplpy, aplpy_kwargs])</code></td>
<td>Use APLpy to make a quick-look image of the projection. This will make the FITSFigure attribute available.</td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>reproject(header[, order])</code></td>
<td>Reproject the image into a new header.</td>
</tr>
<tr>
<td><code>reshape(shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>resize(new_shape[, refcheck])</code></td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td><code>round([decimals, out])</code></td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td><code>searchsorted(v[, side, sorter])</code></td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td><code>setfield(val, dtype[, offset])</code></td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td><code>setflags([write, align, uic])</code></td>
<td>Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.</td>
</tr>
<tr>
<td><code>shrink_mask()</code></td>
<td>Copy of the numpy masked_array shrink_mask method.</td>
</tr>
<tr>
<td><code>sort([axis, kind, order])</code></td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td><code>squeeze([axis])</code></td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td><code>std([axis, dtype, out, ddof, keepdims])</code></td>
<td>Returns the standard deviation of the array elements along given axis.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 13 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subimage(xlo, xhi, ylo, yhi)</code></td>
<td>Extract a region spatially.</td>
</tr>
<tr>
<td><code>sum([axis, dtype, out, keepdims])</code></td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td><code>take(indices[, axis, out, mode])</code></td>
<td>Return an array formed from the elements of a at the given indices.</td>
</tr>
<tr>
<td><code>to(unit[, equivalencies, freq])</code></td>
<td>Return a new Projection of the same class with the specified unit.</td>
</tr>
<tr>
<td><code>to_string([unit, precision, format, subfmt])</code></td>
<td>Generate a string representation of the quantity and its unit.</td>
</tr>
<tr>
<td><code>to_value([unit, equivalencies])</code></td>
<td>The numerical value, possibly in a different unit.</td>
</tr>
<tr>
<td><code>tobytes([order])</code></td>
<td>Construct Python bytes containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>tofile(fid[, sep, format])</code></td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td><code>tostring([order])</code></td>
<td>Construct Python bytes containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>trace([offset, axis1, axis2, dtype, out])</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>var([axis, dtype, out, ddof, keepdims])</code></td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td><code>view([dtype, type])</code></td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td><code>with_beam(beam)</code></td>
<td>Attach a new beam object to the Projection.</td>
</tr>
<tr>
<td><code>with_fill_value(fill_value)</code></td>
<td>Create a new Projection or Slice with a different fill_value.</td>
</tr>
<tr>
<td><code>world_spines()</code></td>
<td>Returns a list of 1D arrays, for the world coordinates along each pixel axis.</td>
</tr>
<tr>
<td><code>write(filename[, format, overwrite])</code></td>
<td>Write the lower dimensional object to a file.</td>
</tr>
</tbody>
</table>

**Attributes Documentation**

T

Same as self.transpose(), except that self is returned if self.ndim < 2.

**Examples**

```python
>>> 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.])
```

d**base**

Base object if memory is from some other object.
**Examples**

The base of an array that owns its memory is None:

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```python
>>> 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):

**_ctypes.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]`.

Note that unlike data_as, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

**_ctypes.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 `ctypes.c_int`, `ctypes.c_long`, or `ctypes.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.
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

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
<ctypes.LP_c_longlong object at 0x01F01300>
```
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.

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 WRITEBACKIFCOPY, 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.
- WRITEBACKIFCOPY 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.

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.

**WRITEBACKIFCOPY (X)**
This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

**UPDATEIFCOPY (U)**
(Deprecated, use WRITEBACKIFCOPY) 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.

**imag**
The imaginary part of the array.

### Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

**info**
Container for meta information like name, description, format. This is required when the object is used as a mixin column within a table, but can be used as a general way to store meta information.
**itemsize**

Length of one array element in bytes.

**Examples**

```python
>>> 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**

```python
>>> 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**

```python
>>> 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.

**See also:**

- `numpy.real`
  
equivalent function

**Examples**
shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape
the array in-place by assigning a tuple of array dimensions to it. As with numpy.reshape, one of the new
shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining
dimensions. Reshaping an array in-place will fail if a copy is required.

See also:

numpy.reshape
    similar function
ndarray.reshape
    similar method

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

>>> np.zeros((4,2))[:,:2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: incompatible shape for a non-contiguous array

size

Number of elements in the array.

Equal to np.prod(a.shape), i.e., the product of the array’s dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods
of obtaining the same value (like the suggested np.prod(a.shape), which returns an instance of np.
int_), and may be relevant if the value is used further in calculations that may overflow a fixed size integer
type.
Examples

```python
>>> 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:

```python
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.

See also:

- `numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```python
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)`.

Examples

```python
>>> 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

Methods Documentation

**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** *(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** *(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** *(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** *(kth, axis=-1, kind='introselect', order=None)*

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

**See also:**
numpy.argpartition
  equivalent function

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(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.

 Raises
**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use \(a\texttt{.real}\) \(\texttt{astype(t)}\).

**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.

**Examples**

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

**byteswap** *(inplace=False)*

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**

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']

>>> A.byteswap(inplace=True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```
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(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(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()
   Complex-conjugate all elements.
   Refer to numpy.conjugate for full documentation.

   See also:

   numpy.conjugate
equivalent function

conjugate()
   Return the complex conjugate, element-wise.
   Refer to numpy.conjugate for full documentation.

   See also:

   numpy.conjugate
equivalent function

convolve_to(beam, convolve=<function convolve_fft>)
   Convolve the image to a specified beam.

   Parameters

   beam
      [radio_beam(Beam] The beam to convolve to
convolve

[function] The astropy convolution function to use, either `astropy.convolution.convolve` or `astropy.convolution.convolve_fft`

Returns

proj

[Projection] A Projection convolved to the given beam object.

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 `numpy.copy()` are very similar, but have different default values for their order= arguments.)

See also:

`numpy.copy`, `numpy.copyto`

Examples

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

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

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

`numpy.cumsum`

equivalent function

decompose

Generates a new `Quantity` with the units decomposed. Decomposed units have only irreducible units in them (see `astropy.units.UnitBase.decompose`).

Parameters

bases

[sequence of UnitBase, optional] The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a `UnitsError` if it’s not possible to do so.

Returns

newq

[Quantity] A new object equal to this quantity with units decomposed.

diagonal

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 `numpy.diagonal()` for full documentation.

See also:

`numpy.diagonal`

equivalent function

diff

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See also:

`numpy.dot`

equivalent function

Examples

```python
>>> 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:

```python
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

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()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

ediff1d(to_end=None, to_begin=None)

fill(value)
Fill the array with a scalar value.

Parameters

value
[scalar] All elements of a will be assigned this value.

Examples

```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])
```

filled(fill_value=None)

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:

- \( \text{ravel} \)
  Return a flattened array.
- flat
  A 1-D flat iterator over the array.

Examples

```python
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

static \texttt{from\_hdu(\texttt{hdu})}

Return a projection from a FITS HDU.

\texttt{getfield(\texttt{dtype}, \texttt{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

```python
>>> 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:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
       [ 0.,  4.]])
```

insert(obj, values, axis=None)

Insert values along the given axis before the given indices and return a new `Quantity` object.

This is a thin wrapper around the `numpy.insert` function.

**Parameters**

- `obj` [int, slice or sequence of ints] Object that defines the index or indices before which `values` is inserted.

- `values` [array-like] Values to insert. If the type of `values` is different from that of quantity, `values` is converted to the matching type. `values` should be shaped so that it can be broadcast appropriately. The unit of `values` must be consistent with this quantity.

- `axis` [int, optional] Axis along which to insert `values`. If `axis` is None then the quantity array is flattened before insertion.

**Returns**

- `out` [Quantity] A copy of quantity with `values` inserted. Note that the insertion does not occur in-place: a new quantity array is returned.

**Examples**

```python
>>> import astropy.units as u

>>> q = [1, 2] * u.m
>>> q.insert(0, 50 * u.cm)
<Quantity [ 0.5, 1., 2.] m>

>>> q = [[1, 2], [3, 4]] * u.m
>>> q.insert(1, [10, 20] * u.m, axis=0)
<Quantity [[ 1., 2.],
          [ 10., 20.],
          [ 3., 4.]] m>

>>> q.insert(1, 10 * u.m, axis=1)
<Quantity [[ 1., 10., 2.],
          [ 3., 10., 4.]] m>
```

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

```python
>>> 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(*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

```python
>>> 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` *(axis=None, out=None, keepdims=False)*

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See also:

`numpy.amax`

equivalent function

`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` *(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

`nansum` *(axis=None, out=None, keepdims=False)*


newbyteorder(new_order='S')

Return the array with the same data viewed with a different byte order.

Equivalent to:

```python
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()

Return the indices of the elements that are non-zero.

Refer to numpy.nonzero for full documentation.

**See also:**

- numpy.nonzero
equivalent function

partition(kth, axis=-1, kind='introselect', order=None)

Rearranges the elements in the array 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.

New in version 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 of 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 to 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

```python
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
array([2, 1, 3, 4])
```

```python
>>> a.partition((1, 3))
array([1, 2, 3, 4])
```

**prod**

Refer to numpy.prod for full documentation.

See also:

numpy.prod
equivalent function

**ptp**

Peak to peak (maximum - minimum) value along a given axis.

Refer to numpy.ptp for full documentation.

See also:

numpy.ptp
equivalent function
**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

**quicklook** *(filename=None, use_aplpy=True, aplpy_kwargs={})*

Use APLpy to make a quick-look image of the projection. This will make the `FITSFigure` attribute available.

If there are unmatched celestial axes, this will instead show an image without axis labels.

Parameters

`filename`

[str or Non] Optional - the filename to save the quicklook to.

**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** *(repeats, axis=None)*

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See also:

`numpy.repeat`

equivalent function

**reproject** *(header, order='bilinear')*

Reproject the image into a new header.

Parameters

`header`

[astropy.io.fits.Header] A header specifying a cube in valid WCS

`order`

[int or str, optional] The order of the interpolation (if `mode` is set to 'interpolation'). This can be either one of the following strings:

• 'nearest-neighbor'

• 'bilinear'
• ‘biquadratic’
• ‘bicubic’

or an integer. A value of 0 indicates nearest neighbor interpolation.

`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

Notes

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11)).`

`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. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

- `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:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
array([[0],
       [1]])
```

```python
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> 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...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ... ValueError: cannot resize an array that has been referenced ...
```

Unless refcheck is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

**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**(*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**(*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**

```python
>>> 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**(*write=None, align=None, uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY 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 WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags 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 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, 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) (deprecated), replaced by WRITEBACKIFCOPY;
- WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

**Examples**

```python
>>> 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
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
```

(continues on next page)
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False

```python
>>> y.setflags(uic=1)
```

Traceback (most recent call last):
  File "<stdin>"}, line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True

**shrink_mask()**

Copy of the numpy masked_array shrink_mask method. This is essentially a hack needed for matplotlib to show images.

**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', 'stable'}, 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

```python
>>> 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:

```python
>>> 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", '<i8')])
```

`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(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

`subimage(xlo='min', xhi='max', ylo='min', yhi='max')`

Extract a region spatially.

Parameters

`[xy]lo/[xy]hi`

[int or astropy.units.Quantity or min/max] The endpoints to extract. If given as a quantity, will be interpreted as World coordinates. If given as a string or int, will be interpreted as pixel coordinates.

`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
\texttt{swapaxes}(axis1, axis2)
Return a view of the array with axis1 and axis2 interchanged.
Refer to \texttt{numpy.swapaxes} for full documentation.

See also:
\texttt{numpy.swapaxes}
equivalent function

\texttt{take}(indices, axis=None, out=None, mode='raise')
Return an array formed from the elements of \texttt{a} at the given indices.
Refer to \texttt{numpy.take} for full documentation.

See also:
\texttt{numpy.take}
equivalent function

\texttt{to}(unit, equivalencies=[], freq=None)
Return a new \texttt{Projection} of the same class with the specified unit.
See \texttt{astropy.units.Quantity.to} for further details.

\texttt{to_string}(unit=None, precision=None, format=None, subfmt=None)
Generate a string representation of the quantity and its unit.
The behavior of this function can be altered via the \texttt{numpy.set_printoptions} function and its various
keywords. The exception to this is the \texttt{threshold} keyword, which is controlled via the [\texttt{units.quantity}] configuration item \texttt{latex_array_threshold}. This is treated separately because the numpy default of 1000 is too big for most browsers to handle.

Parameters

\texttt{unit}
[\texttt{UnitBase}, optional] Specifies the unit. If not provided, the unit used to initialize the quantity will be used.

\texttt{precision}
[\texttt{numeric}, optional] The level of decimal precision. If \texttt{None}, or not provided, it will be determined from NumPy print options.

\texttt{format}
[\texttt{str}, optional] The format of the result. If not provided, an unadorned string is returned. Supported values are:
- \texttt{‘latex’}: Return a LaTeX-formatted string

\texttt{subfmt}
[\texttt{str}, optional] Subformat of the result. For the moment, only used for \texttt{format=’latex’}. Supported values are:
- \texttt{‘inline’}: Use $ \ldots $ as delimiters.
- \texttt{‘display’}: Use $\displaystyle \ldots $ as delimiters.

Returns
A string with the contents of this Quantity

to_value(unit=None, equivalencies=[])
The numerical value, possibly in a different unit.

Parameters

unit
[UnitBase instance or str, optional] The unit in which the value should be given. If not given or None, use the current unit.

equivalencies
[list of equivalence pairs, optional] A list of equivalence pairs to try if the units are not directly convertible (see Equivalencies). If not provided or [], class default equivalencies will be used (none for Quantity, but may be set for subclasses). If None, no equivalences will be applied at all, not even any set globally or within a context.

Returns

value
[[ndarray or scalar] The value in the units specified. For arrays, this will be a view of the data if no unit conversion was necessary.

See also:

to
Get a new instance in a different unit.

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.

New in version 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

```python
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x00\x03\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
(continues on next page)```
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.

When `fid` is a file object, array contents are directly written to the file, bypassing the file object’s `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

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

```python
>>> 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(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

```python
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
b'\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00\x00\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
'b'\x00\x00\x00\x00\x02\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

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(*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**

```python
>>> 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]])
```

**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**(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**

```python
>>> x = np.array([[1, 2]], dtype=[('a', np.int8), ('b', np.int8)])

Viewing array data using a different type and dtype:

```python
def moiddle
```np
```int16`<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

```python
>>> x = np.array([[1, 2], [3, 4]], dtype=[('a', np.int8), ('b', np.int8)])
```moiddle
>>> x = x.view(dtype=np.int8).reshape(-1,2)
```>>> x
```array([[1, 2],
```[3, 4]], dtype=int8)
```>>> x.mean(0)
```array([ 2., 3.])
```Making changes to the view changes the underlying array

```python
>>> x[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')])
```

**with_beam(beam)**

Attach a new beam object to the Projection.

**Parameters**

- **beam**
  

**with_fill_value(fill_value)**

Create a new `Projection` or `Slice` with a different `fill_value`.

**world_spines()**

Returns a list of 1D arrays, for the world coordinates along each pixel axis.

Raises error if this operation is ill-posed (e.g. rotated world coordinates, strong distortions)

This method is not currently implemented. Use world() instead.

**write(filename, format=None, overwrite=False)**

Write the lower dimensional object to a file.

**Parameters**

- **filename**
  
  [str] The path to write the file to

- **format**
  
  [str] The kind of file to write. (Currently limited to ‘fits’)

- **overwrite**
  
  [bool] If True, overwrite filename if it exists
Slice

```python
class spectral_cube.Slice
    Bases: spectral_cube.lower_dimensional_structures.Projection
```

Attributes Summary

- `T`: Same as self.transpose(), except that self is returned if self.ndim < 2.
- `base`: Base object if memory is from some other object.
- `ctypes`: An object to simplify the interaction of the array with the ctypes module.
- `data`: Python buffer object pointing to the start of the array's data.
- `dtype`: Data-type of the array's elements.
- `flags`: Information about the memory layout of the array.
- `imag`: The imaginary part of the array.
- `info`: Container for meta information like name, description, format.
- `itemsize`: Length of one array element in bytes.
- `nbytes`: Total bytes consumed by the elements of the array.
- `ndim`: Number of array dimensions.
- `real`: The real part of the array.
- `shape`: Tuple of array dimensions.
- `size`: Number of elements in the array.
- `strides`: Tuple of bytes to step in each dimension when traversing an array.

Methods Summary

- `all([axis, out, keepdims])`: Returns True if all elements evaluate to True.
- `any([axis, out, keepdims])`: Returns True if any of the elements of a evaluate to True.
- `argmax([axis, out])`: Return indices of the maximum values along the given axis.
- `argmin([axis, out])`: Return indices of the minimum values along the given axis of a.
- `argpartition(kth[, axis, kind, order])`: Returns the indices that would partition this array.
- `argsort([axis, kind, order])`: Returns the indices that would sort this array.
- `astype(dtype[, order, casting, subok, copy])`: Copy of the array, cast to a specified type.
- `byteswap([inplace])`: Swap the bytes of the array elements.
- `choose(choices[, out, mode])`: Use an index array to construct a new array from a set of choices.
- `clip([min, max, out])`: Return an array whose values are limited to [min, max].
- `compress(condition[, axis, out])`: Return selected slices of this array along given axis.
- `conj()`: Complex-conjugate all elements.
- `conjugate()`: Return the complex conjugate, element-wise.
- `convolve_to(beam[, convolve])`: Convolve the image to a specified beam.
Table 15 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copy(order)</code></td>
<td>Return a copy of the array.</td>
</tr>
<tr>
<td><code>cumprod(axis, dtype, out)</code></td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td><code>cumsum(axis, dtype, out)</code></td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td><code>decompose(bases)</code></td>
<td>Generates a new <code>Quantity</code> with the units decomposed.</td>
</tr>
<tr>
<td><code>diagonal(offset, axis1, axis2)</code></td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td><code>diff(n, axis)</code></td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td><code>dump(file)</code></td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td><code>dumps()</code></td>
<td>Returns the pickle of the array as a string.</td>
</tr>
<tr>
<td><code>ediff1d(to_end, to_begin)</code></td>
<td>Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.</td>
</tr>
<tr>
<td><code>fill(value)</code></td>
<td>Fill the array with a scalar value.</td>
</tr>
<tr>
<td><code>filled(fill_value)</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>from_hdu(hdu)</code></td>
<td>Return a projection from a FITS HDU.</td>
</tr>
<tr>
<td><code>getfield(dtype[, offset])</code></td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td><code>insert(obj, values[, axis])</code></td>
<td>Insert values along the given axis before the given indices and return a new <code>Quantity</code> object.</td>
</tr>
<tr>
<td><code>item(*args)</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>itemset(*args)</code></td>
<td>Insert scalar into an array (scalar is cast to array’s dtype, if possible)</td>
</tr>
<tr>
<td><code>max(axis, out, keepdims)</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>mean(axis, dtype, out, keepdims)</code></td>
<td>Returns the average of the array elements along given axis.</td>
</tr>
<tr>
<td><code>min(axis, out, keepdims)</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>nansum(axis, out, keepdims)</code></td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>newbyteorder(new_order)</code></td>
<td>Return the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>partition(kth[, axis, kind, order])</code></td>
<td>Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.</td>
</tr>
<tr>
<td><code>prod(axis, dtype, out, keepdims)</code></td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
<tr>
<td><code>ptp(axis, out, keepdims)</code></td>
<td>Set a.flat[n] = values[n] for all n in indices.</td>
</tr>
<tr>
<td><code>quicklook([filename, use_aplpy, aplpy_kwargs])</code></td>
<td>Use APLPy to make a quick-look image of the projection. This will make the FITSFigure attribute available.</td>
</tr>
<tr>
<td><code>ravel(order)</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>reproject(header[, order])</code></td>
<td>Reproject the image into a new header.</td>
</tr>
<tr>
<td><code>reshape(shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>resize(new_shape[, refcheck])</code></td>
<td>Change shape and size of array in-place.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 15 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>round([decimals, out])</code></td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td><code>searchsorted(v[, side, sorter])</code></td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td><code>setfield(val, dtype[, offset])</code></td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td><code>setflags([write, align, uic])</code></td>
<td>Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.</td>
</tr>
<tr>
<td><code>shrink_mask()</code></td>
<td>Copy of the numpy masked_array shrink_mask method.</td>
</tr>
<tr>
<td><code>sort([axis, kind, order])</code></td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td><code>squeeze([axis])</code></td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td><code>std([axis, dtype, out, ddof, keepdims])</code></td>
<td>Returns the standard deviation of the array elements along given axis.</td>
</tr>
<tr>
<td><code>subimage([xlo, xhi, ylo, yhi])</code></td>
<td>Extract a region spatially.</td>
</tr>
<tr>
<td><code>sum([axis, dtype, out, keepdims])</code></td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td><code>take(indices[, axis, out, mode])</code></td>
<td>Return an array formed from the elements of a at the given indices.</td>
</tr>
<tr>
<td><code>to(unit[, equivalencies, freq])</code></td>
<td>Return a newProjection of the same class with the specified unit.</td>
</tr>
<tr>
<td><code>to_string([unit, precision, format, subfmt])</code></td>
<td>Generate a string representation of the quantity and its unit.</td>
</tr>
<tr>
<td><code>to_value([unit, equivalencies])</code></td>
<td>The numerical value, possibly in a different unit.</td>
</tr>
<tr>
<td><code>tobytes([order])</code></td>
<td>Construct Python bytes containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>tofile(fid[, sep, format])</code></td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td><code>tostring([order])</code></td>
<td>Construct Python bytes containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>trace([offset, axis1, axis2, dtype, out])</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>var([axis, dtype, out, ddof, keepdims])</code></td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td><code>view([dtype, type])</code></td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td><code>with_beam(beam)</code></td>
<td>Attach a new beam object to the Projection.</td>
</tr>
<tr>
<td><code>with_fill_value(fill_value)</code></td>
<td>Create a newProjection or Slice with a different fill_value.</td>
</tr>
<tr>
<td><code>world_spines()</code></td>
<td>Returns a list of 1D arrays, for the world coordinates along each pixel axis.</td>
</tr>
<tr>
<td><code>write(filename[, format, overwrite])</code></td>
<td>Write the lower dimensional object to a file.</td>
</tr>
</tbody>
</table>

**Attributes Documentation**

\[T\]

Same as self.transpose(), except that self is returned if self.ndim < 2.
Examples

```python
>>> 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.])
```

base

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is None:

```python
>>> x = np.array([1,2,3,4])
>>> x.base
is None
True
```

Slicing creates a view, whose memory is shared with x:

```python
>>> 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):

.ctypes.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].

Note that unlike data_as, a reference will not be kept to the array: code like ctypes.c_void_p((a + b).ctypes.data) will result in a pointer to a deallocated array, and should be spelled (a + b).ctypes.data_as(ctypes.c_void_p)

.ctypes.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 basetype could be ctypes.c_int, ctypes.c_long, or ctypes.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.

.ctypes.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.

.ctypes.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)).

The returned pointer will keep a reference to the array.

.ctypes.shape_as(obj)
Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

.ctypes.strides_as(obj)
Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

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

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
```

(continues on next page)
<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents</code></td>
<td><code>c_long(0)</code></td>
</tr>
<tr>
<td><code>x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents</code></td>
<td><code>c_longlong(4294967296L)</code></td>
</tr>
<tr>
<td><code>x.ctypes.shape</code></td>
<td><code>&lt;numpy.core._internal.c_long_Array_2 object at 0x01FFD580&gt;</code></td>
</tr>
<tr>
<td><code>x.ctypes.shape_as(ctypes.c_long)</code></td>
<td><code>&lt;numpy.core._internal.c_long_Array_2 object at 0x01FCE620&gt;</code></td>
</tr>
<tr>
<td><code>x.ctypes.strides</code></td>
<td><code>&lt;numpy.core._internal.c_long_Array_2 object at 0x01FCE620&gt;</code></td>
</tr>
<tr>
<td><code>x.ctypes.strides_as(ctypes.c_longlong)</code></td>
<td><code>&lt;numpy.core._internal.c_longlong_Array_2 object at 0x01F01300&gt;</code></td>
</tr>
</tbody>
</table>

**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**

```python
>>> 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.

**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 WRITEBACKIFCOPY, 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 to `False`.
- **WRITEBACKIFCOPY** can only be set to `False`.
- **ALIGNED** can only be set to `True` if the data is truly aligned.
- **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.

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.

**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.

**WRITEBACKIFCOPY (X)**

This array is a copy of some other array. The C-API function `PyArray_ResolveWritebackIfCopy` must be called before deallocating to the base array will be updated with the contents of this array.

**UPDATEIFCOPY (U)**

(Deprecated, use WRITEBACKIFCOPY) 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.

imag
The imaginary part of the array.

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
```

info
Container for meta information like name, description, format. This is required when the object is used as a mixin column within a table, but can be used as a general way to store meta information.

Examples

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
```

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

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
```

ndim
Number of array dimensions.
Examples

```python
>>> 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.

See also:

- `numpy.real`
  equivalent function

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

**shape**

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with `numpy.reshape`, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

See also:

- `numpy.reshape`
  similar function
- `ndarray.reshape`
  similar method

Examples

```python
>>> 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.]],
   [[ 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)
```
size

Number of elements in the array.

Equal to np.prod(a.shape), i.e., the product of the array’s dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested np.prod(a.shape), which returns an instance of np.int_), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```python
>>> 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.

See also:

numpy.lib.stride_tricks.as_strided

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```python
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).
Examples

```python
>>> 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
```

Methods Documentation

all(\textit{axis=\texttt{None}, out=\texttt{None}, keepdims=\texttt{False}})

Returns True if all elements evaluate to True.

Refer to \texttt{numpy.all} for full documentation.

See also:

\texttt{numpy.all}

\texttt{equivalent function}

any(\textit{axis=\texttt{None}, out=\texttt{None}, keepdims=\texttt{False}})

Returns True if any of the elements of a evaluate to True.

Refer to \texttt{numpy.any} for full documentation.

See also:

\texttt{numpy.any}

\texttt{equivalent function}

argmax(\textit{axis=\texttt{None}, out=\texttt{None}})

Return indices of the maximum values along the given axis.

Refer to \texttt{numpy.argmax} for full documentation.

See also:
numpy.argmax
   equivalent function

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(kth, axis=-1, kind='introselect', order=None)
   Returns the indices that would partition this array.
   Refer to numpy.argpartition for full documentation.
   New in version 1.8.0.

See also:

numpy.argpartition
   equivalent function

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(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
   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 cast-
   ing 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
[narray] 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.

Raises

ComplexWarning
When casting from complex to float or int. To avoid this, one should use a.real.
astype(t).

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.

Examples

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

byteswap(inplace=False)
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

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

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(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(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()

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See also:
**numpy.conjugate**

Equivalent function

**conjugate()**

Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See also:

**numpy.conjugate**

Equivalent function

**convolve_to(beam, convolve=<function convolve_fft>)**

Convolve the image to a specified beam.

**Parameters**

- `beam` ([radio_beam.Beam]) The beam to convolve to

- `convolve` ([function]) The astropy convolution function to use, either `astropy.convolution.convolve` or `astropy.convolution.convolve_fft`

**Returns**

- `proj` ([Projection]) A Projection convolved to the given beam object.

**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 `numpy.copy()` are very similar, but have different default values for their order= arguments.)

See also:

`numpy.copy`, `numpy.copyto`

**Examples**

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)
```
```python
>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

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(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
decompose(bases=[])
Generates a new Quantity with the units decomposed. Decomposed units have only irreducible units in them (see astropy.units.UnitBase.decompose).

Parameters

bases
[sequence of UnitBase, optional] The bases to decompose into. When not provided, decomposes down to any irreducible units. When provided, the decomposed result will only contain the given units. This will raise a UnitsError if it's not possible to do so.

Returns

newq
[Quantity] A new object equal to this quantity with units decomposed.
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 numpy.diagonal() for full documentation.

See also:

numpy.diagonal
equivalent function
\texttt{diff}(n=1, \texttt{axis}=-1)

\texttt{dot}(b, \texttt{out}=None)

Dot product of two arrays.

Refer to \texttt{numpy.dot} for full documentation.

See also:

\texttt{numpy.dot}

equivalent function

**Examples**

```python
>>> 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:

```python
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump(\texttt{file})

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

\texttt{file}

[str] A string naming the dump file.

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

ediff1d(\texttt{to_end}=None, \texttt{to_begin}=None)

fill(\texttt{value})

Fill the array with a scalar value.

**Parameters**

\texttt{value}

[scalar] All elements of \texttt{a} will be assigned this value.
Examples

```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```

documented

The `filled` function has two additional arguments:

- `fill_value` - sets the value to fill the array with. If not specified, it defaults to `None`.
- `order` - specifies the order in which the elements are flattened. Options are `'C'` for row-major order, `'F'` for column-major order, `'A'` for column-major if the array is Fortran contiguous, and `'K'` to flatten in the order the elements occur in memory.

**Parameters**

- `order` - specifies the order of flattening. Options are `'C'` for row-major, `'F'` for column-major, `'A'` for column-major if the array is Fortran contiguous, and `'K'` for the order of memory access.

**Returns**

- `y` - a flattened copy of the input array.

**See also:**

- `ravel` - returns a flattened array.
- `flat` - a 1-D flat iterator.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

The `static from_hdu` function:

- Returns a projection from a FITS HDU.

The `getfield` function:

- 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

```python
g = np.diag([1.+1.j]*2)
g[1, 1] = 2 + 4.j
print(g)
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
g.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:

```python
g.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```

**insert(obj, values, axis=None)**

Insert values along the given axis before the given indices and return a new `Quantity` object.

This is a thin wrapper around the `numpy.insert` function.

Parameters

- **obj**
  
  [int, slice or sequence of ints] Object that defines the index or indices before which values is inserted.

- **values**
  
  [array-like] Values to insert. If the type of values is different from that of quantity, values is converted to the matching type. values should be shaped so that it can be broadcast appropriately. The unit of values must be consistent with this quantity.

- **axis**
  
  [int, optional] Axis along which to insert values. If axis is None then the quantity array is flattened before insertion.

Returns

- **out**
  
  [Quantity] A copy of quantity with values inserted. Note that the insertion does not occur in-place: a new quantity array is returned.
Examples

```python
>>> import astropy.units as u

>>> q = [1, 2] * u.m
>>> q.insert(0, 50 * u.cm)
<Quantity [ 0.5, 1., 2.] m>

>>> q = [[1, 2], [3, 4]] * u.m
>>> q.insert(1, [10, 20] * u.m, axis=0)
<Quantity [[ 1., 2.],
          [10., 20.],
          [ 3., 4.]] m>

>>> q.insert(1, 10 * u.m, axis=1)
<Quantity [[ 1., 10., 2.],
          [ 3., 10., 4.]] m>
```

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
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.item(3)  # 2
>>> x.item(7)  # 5

>>> x.item((0, 1))  # 1
>>> x.item((2, 2))  # 3

max(axis=None, out=None, keepdims=False)
Return the maximum along a given axis.

Refer to numpu.amax for full documentation.

See also:
numpy.amax
    equivalent function

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(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

nansum(axis=None, out=None, keepdims=False)

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()
    Return the indices of the elements that are non-zero.
    Refer to numpy.nonzero for full documentation.
    See also:
    
    numpy.nonzero
    equivalent function

partition(kth, axis=-1, kind='introselect', order=None)
    Rearranges the elements in the array 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.
    New in version 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 of 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 to 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 parititioned copy of an array.
    argpartition
        Indirect partition.
    sort
        Full sort.
    Notes
    See np.partition for notes on the different algorithms.
Examples

```python
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])
```

`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(axis=None, out=None, keepdims=False)`

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also:

- `numpy.ptp` equivalent function

`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

`quicklook(filename=None, use_aplpy=True, aplpy_kwargs={})`

Use APLpy to make a quick-look image of the projection. This will make the FITSFigure attribute available.

If there are unmatched celestial axes, this will instead show an image without axis labels.

Parameters:

- `filename`
  - [str or None] Optional - the filename to save the quicklook to.

`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**(repeats, axis=None)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

**numpy.repeat**

Equivalent function

**reproject**(header, order='bilinear')

Reproject the image into a new header.

**Parameters**

**header**

`astropy.io.fits.Header` A header specifying a cube in valid WCS

**order**

[int or str, optional] The order of the interpolation (if mode is set to 'interpolation').

This can be either one of the following strings:

- 'nearest-neighbor'
- 'bilinear'
- 'biquadratic'
- 'bicubic'

or an integer. A value of 0 indicates nearest neighbor interpolation.

**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

**Notes**

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape((10, 11))` is equivalent to `a.reshape((10, 11))`.

**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. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

**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:

```python
gbf>> a = np.array([[0, 1], [2, 3]], order='C')
gbf>> a.resize((2, 1))
gbf>> a
gbfarray([[0],
        [1]])
```

```python
gbf>> a = np.array([[0, 1], [2, 3]], order='F')
gbf>> a.resize((2, 1))
gbf>> a
gbfarray([[0],
        [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:
```python
>>> 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...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...  
ValueError: cannot resize an array that has been referenced ...
```

Unless refcheck is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

```
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(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(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:

gffield

Examples

```python
>>> 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(write=None, align=None, uic=None)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY 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 WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags 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 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UP-
DATEIFCOPY, 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) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIFCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```python
given_data = np.array([[3, 1, 7], [2, 0, 0], [8, 5, 9]])
given_data.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
given_data.setflags(write=0, align=0)
given_data.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
given_data.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

shrink_mask()
Copy of the numpy masked_array shrink_mask method. This is essentially a hack needed for matplotlib to show images.

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', 'stable'}, optional] Sorting algorithm. Default is 'quicksort'.

2.3. API Documentation 153
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)
array([[1, 4],
       [3, 1]])
>>> a.sort(axis=0)
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')
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])

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\( (\text{axis}=\text{None}, \text{dtype}=\text{None}, \text{out}=\text{None}, \text{ddof}=0, \text{keepdims}=\text{False}) \)

Returns the standard deviation of the array elements along given axis.

Refer to numpy.std for full documentation.

See also:

- numpy.std
  equivalent function

subimage\( (\text{xlo}=\text{min}', \text{xhi}=\text{max}', \text{ylo}=\text{min}', \text{yhi}=\text{max}') \)

Extract a region spatially.

Parameters

\[ \text{[xy]lo/[xy]hi} \]

[int or astropy.units.Quantity or min/max] The endpoints to extract. If given as a quantity, will be interpreted as World coordinates. If given as a string or int, will be interpreted as pixel coordinates.

sum\( (\text{axis}=\text{None}, \text{dtype}=\text{None}, \text{out}=\text{None}, \text{keepdims}=\text{False}) \)

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\( (\text{axis1}, \text{axis2}) \)

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\( (\text{indices}, \text{axis}=\text{None}, \text{out}=\text{None}, \text{mode}=\text{raise}') \)

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

to\( (\text{unit}, \text{equivalencies}=[], \text{freq}=\text{None}) \)

Return a new Projection of the same class with the specified unit.

See astropy.units.Quantity.to for further details.

to_string\( (\text{unit}=\text{None}, \text{precision}=\text{None}, \text{format}=\text{None}, \text{subfmt}=\text{None}) \)

Generate a string representation of the quantity and its unit.

The behavior of this function can be altered via the numpy.set_printoptions function and its various keywords. The exception to this is the threshold keyword, which is controlled via the [units.quantity]
configuration item `latex_array_threshold`. This is treated separately because the numpy default of 1000 is too big for most browsers to handle.

**Parameters**

- **unit**
  - `[UnitBase, optional]` Specifies the unit. If not provided, the unit used to initialize the quantity will be used.

- **precision**
  - `[numeric, optional]` The level of decimal precision. If `None`, or not provided, it will be determined from NumPy print options.

- **format**
  - `[str, optional]` The format of the result. If not provided, an unadorned string is returned. Supported values are:
    - ‘latex’: Return a LaTeX-formatted string

- **subfmt**
  - `[str, optional]` Subformat of the result. For the moment, only used for format=”latex”. Supported values are:
    - ‘inline’: Use $ ... $ as delimiters.
    - ‘display’: Use $\displaystyle ... $ as delimiters.

**Returns**

- **lstr**
  - A string with the contents of this Quantity

**to_value**(unit=`None`, equivalencies=[])  
The numerical value, possibly in a different unit.

**Parameters**

- **unit**
  - `[UnitBase instance or str, optional]` The unit in which the value should be given. If not given or `None`, use the current unit.

- **equivalencies**
  - `[list of equivalence pairs, optional]` A list of equivalence pairs to try if the units are not directly convertible (see `Equivalencies`). If not provided or [], class default equivalencies will be used (none for `Quantity`, but may be set for subclasses). If `None`, no equivalencies will be applied at all, not even any set globally or within a context.

**Returns**

- **value**
  - `[ndarray or scalar]` The value in the units specified. For arrays, this will be a view of the data if no unit conversion was necessary.

**See also:**

- **to**
  - Get a new instance in a different unit.
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.

New in version 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

```python
>>> 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\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\x00'
```

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.
When fid is a file object, array contents are directly written to the file, bypassing the file object’s write method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support fileno() (e.g., BytesIO).

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

```python
>>> 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(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

```python
>>> 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\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\x00\x00'
```

`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(*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**
```python
>>> 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**(*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`

**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 reinter-pretation 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

```python
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])

Viewing array data using a different type and dtype:

```python
>>> 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

```python
>>> 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

```python
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```python
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```python
>>> 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.:}

```python
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, :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')])
```
**with_beam(beam)**

Attach a new beam object to the Projection.

**Parameters**

beam

**with_fill_value(fill_value)**

Create a new Projection or Slice with a different fill_value.

**world_spines()**

Returns a list of 1D arrays, for the world coordinates along each pixel axis.

Raises error if this operation is ill-posed (e.g. rotated world coordinates, strong distortions)

This method is not currently implemented. Use world() instead.

**write(filename, format=None, overwrite=False)**

Write the lower dimensional object to a file.

**Parameters**

filename
[str] The path to write the file to

format
[st] The kind of file to write. (Currently limited to ‘fits’)

overwrite
[bool] If True, overwrite filename if it exists

---

**SpectralCube**

class spectral_cube.SpectralCube(data, wcs, mask=None, meta=None, fill_value=nan, header=None, allow_huge_operations=False, beam=None, wcs_tolerance=0.0, **kwargs)


**BeamMixinClass**

**Methods Summary**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply_function(function[, axis, weights, ...])</td>
<td>Apply a function to valid data along the specified axis or to the whole cube, optionally using a weight array that is the same shape (or at least can be sliced in the same way)</td>
</tr>
<tr>
<td>apply_function_parallel_spatial(function[, ...])</td>
<td>Apply a function in parallel along the spatial dimension.</td>
</tr>
<tr>
<td>apply_function_parallel_spectral(function[, ...])</td>
<td>Apply a function in parallel along the spectral dimension.</td>
</tr>
<tr>
<td>apply_numpy_function(function[, fill, ...])</td>
<td>Apply a numpy function to the cube</td>
</tr>
<tr>
<td>argmax((axis, how))</td>
<td>Return the index of the maximum data value.</td>
</tr>
<tr>
<td>argmin((axis, how))</td>
<td>Return the index of the minimum data value.</td>
</tr>
<tr>
<td>chunked([chunksize])</td>
<td>Not Implemented.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 16 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closest_spectral_channel(value)</td>
<td>Find the index of the closest spectral channel to the specified spectral coordinate.</td>
</tr>
<tr>
<td>convolve_to(beam[, convolve, update_function])</td>
<td>Convolve each channel in the cube to a specified beam</td>
</tr>
<tr>
<td>downsample_axis(factor, axis[, estimator, ...])</td>
<td>Downsample the cube by averaging over <code>factor</code> pixels along an axis.</td>
</tr>
<tr>
<td>filled(fill_value)</td>
<td></td>
</tr>
<tr>
<td>find_lines([velocity_offset, ...])</td>
<td>Using astroquery's splatalogue interface, search for lines within the spectral band.</td>
</tr>
<tr>
<td>flattened([slice, weights])</td>
<td>Return a slice of the cube giving only the valid data (i.e., removing bad values)</td>
</tr>
<tr>
<td>flattened_world([view])</td>
<td>Retrieve the world coordinates corresponding to the extracted flattened version of the cube</td>
</tr>
<tr>
<td>get_mask_array()</td>
<td>Convert the mask to a boolean numpy array</td>
</tr>
<tr>
<td>linewidth_fwhm([how])</td>
<td>Compute a (FWHM) linewidth map along the spectral axis.</td>
</tr>
<tr>
<td>linewidth_sigma([how])</td>
<td>Compute a (sigma) linewidth map along the spectral axis.</td>
</tr>
<tr>
<td>mad_std([axis, how])</td>
<td>Use astropy's mad_std to compute the standard deviation</td>
</tr>
<tr>
<td>mask_channels(goodchannels)</td>
<td>Helper function to mask out channels.</td>
</tr>
<tr>
<td>max([axis, how])</td>
<td>Return the maximum data value of the cube, optionally over an axis.</td>
</tr>
<tr>
<td>mean([axis, how])</td>
<td>Return the mean of the cube, optionally over an axis.</td>
</tr>
<tr>
<td>median([axis, iterate_rays])</td>
<td>Compute the median of an array, optionally along an axis.</td>
</tr>
<tr>
<td>min([axis, how])</td>
<td>Return the minimum data value of the cube, optionally over an axis.</td>
</tr>
<tr>
<td>minimal_subcube([spatial_only])</td>
<td>Return the minimum enclosing subcube where the mask is valid</td>
</tr>
<tr>
<td>moment([order, axis, how])</td>
<td>Compute moments along the spectral axis.</td>
</tr>
<tr>
<td>moment0([axis, how])</td>
<td>Compute the zeroth moment along an axis.</td>
</tr>
<tr>
<td>moment1([axis, how])</td>
<td>Compute the 1st moment along an axis.</td>
</tr>
<tr>
<td>moment2([axis, how])</td>
<td>Compute the 2nd moment along an axis.</td>
</tr>
<tr>
<td>percentile(q[, axis, iterate_rays])</td>
<td>Return percentiles of the data.</td>
</tr>
<tr>
<td>read(filename[, format, hdu])</td>
<td>Read a spectral cube from a file.</td>
</tr>
<tr>
<td>reproject(header[, order, use_memmap, filled])</td>
<td>Spatially reproject the cube into a new header.</td>
</tr>
<tr>
<td>sigma_clip(threshold[, verbose, use_memmap, ...])</td>
<td>Run astropy’s sigma clipper, converting all bad values to NaN.</td>
</tr>
<tr>
<td>spatial_smooth(kernel[, convolve])</td>
<td>Smooth the image in each spatial-spatial plane of the cube.</td>
</tr>
<tr>
<td>spatial_smooth_median(ksize[, update_function])</td>
<td>Smooth the image in each spatial-spatial plane of the cube using a median filter.</td>
</tr>
<tr>
<td>spectral_interpolate(spectral_grid[, ...])</td>
<td>Resample the cube spectrally onto a specific grid</td>
</tr>
<tr>
<td>spectral_slab(lo, hi)</td>
<td>Extract a new cube between two spectral coordinates</td>
</tr>
<tr>
<td>spectral_smooth(kernel[, convolve, verbose, ...])</td>
<td>Smooth the cube along the spectral dimension</td>
</tr>
<tr>
<td>spectral_smooth_median(ksize[, use_memmap, ...])</td>
<td>Smooth the cube along the spectral dimension</td>
</tr>
<tr>
<td>std([axis, how, ddof])</td>
<td>Return the standard deviation of the cube, optionally over an axis.</td>
</tr>
<tr>
<td>subcube([xlo, xhi, ylo, yhi, zlo, zhi, ...])</td>
<td>Extract a sub-cube spatially and spectrally.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 16 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subcube_from_crtfregion(crtf_region[, ...])</code></td>
<td>Extract a masked subcube from a CRTF region.</td>
</tr>
<tr>
<td><code>subcube_from_ds9region(ds9_region[, allow_empty])</code></td>
<td>Extract a masked subcube from a ds9 region (only functions on celestial dimensions).</td>
</tr>
<tr>
<td><code>subcube_from_mask(region_mask)</code></td>
<td>Given a mask, return the minimal subcube that encloses the mask.</td>
</tr>
<tr>
<td><code>subcube_from_regions(region_list[, allow_empty])</code></td>
<td>Extract a masked subcube from a list of regions. Region object (only functions on celestial dimensions).</td>
</tr>
<tr>
<td><code>subcube_slices_from_mask(region_mask[, ...])</code></td>
<td>Given a mask, return the slices corresponding to the minimum subcube that encloses the mask.</td>
</tr>
<tr>
<td><code>sum([axis, how])</code></td>
<td>Return the sum of the cube, optionally over an axis.</td>
</tr>
<tr>
<td><code>to(unit[, equivalencies])</code></td>
<td>Return the cube converted to the given unit (assuming it is equivalent).</td>
</tr>
<tr>
<td><code>to_ds9([ds9id, newframe])</code></td>
<td>Send the data to ds9 (this will create a copy in memory).</td>
</tr>
<tr>
<td><code>to_glue([name, glue_app, dataset, start_gui])</code></td>
<td>Send data to a new or existing Glue application.</td>
</tr>
<tr>
<td><code>to_pvextractor()</code></td>
<td>Open the cube in a quick viewer written in matplotlib that allows you to create PV extractions within the GUI.</td>
</tr>
<tr>
<td><code>to_yt([spectral_factor, nprocs])</code></td>
<td>Convert a spectral cube to a yt object that can be further analyzed in yt.</td>
</tr>
<tr>
<td><code>unmasked_copy()</code></td>
<td>Return a copy of the cube with no mask (i.e., all data included).</td>
</tr>
<tr>
<td><code>with_beam(beam)</code></td>
<td>Attach a beam object to the SpectralCube.</td>
</tr>
<tr>
<td><code>with_fill_value(fill_value)</code></td>
<td>Create a new object with a different fill_value.</td>
</tr>
<tr>
<td><code>with_mask(mask[, inherit_mask, wcs_tolerance])</code></td>
<td>Return a new SpectralCube instance that contains a composite mask of the current SpectralCube and the new mask.</td>
</tr>
<tr>
<td><code>with_spectral_unit(unit[, ...])</code></td>
<td>Returns a new Cube with a different Spectral Axis unit.</td>
</tr>
<tr>
<td><code>world_spines()</code></td>
<td>Returns a list of 1D arrays, for the world coordinates along each pixel axis.</td>
</tr>
<tr>
<td><code>write(filename[, overwrite, format])</code></td>
<td>Write the spectral cube to a file.</td>
</tr>
</tbody>
</table>

Methods Documentation

`apply_function(function, axis=None, weights=None, unit=None, projection=False, progressbar=False, update_function=None, keep_shape=False, **kwargs)`

Apply a function to valid data along the specified axis or to the whole cube, optionally using a weight array that is the same shape (or at least can be sliced in the same way).

Parameters

function

[function] A function that can be applied to a numpy array. Does not need to be nan-aware.

axis

[1, 2, 3, or None] The axis to operate along. If None, the return is scalar.

weights

[(optional) np.ndarray] An array with the same shape (or slicing abilities/results) as the data cube.

unit
spectralcube Documentation, Release 0.4.5.dev1105

[(optional) Unit] The unit of the output projection or value. Not all functions should return quantities with units.

projection
[bool] Return a projection if the resulting array is 2D?

progressbar
[bool] Show a progressbar while iterating over the slices/rays through the cube?

keep_shape
[bool] If True, the returned object will be the same dimensionality as the cube.

update_function
[function] An alternative tracker for the progress of applying the function to the cube data. If progressbar is True, this argument is ignored.

Returns

result
[Projection or Quantity or float] The result depends on the value of axis, projection, and unit. If axis is None, the return will be a scalar with or without units. If axis is an integer, the return will be a Projection if projection is set.

apply_function_parallel_spatial(function, num_cores=None, verbose=0, use_memmap=True, parallel=True, **kwargs)
Apply a function in parallel along the spatial dimension. The function will be performed on data with masked values replaced with the cube’s fill value.

Parameters

function
[function] The function to apply in the spatial dimension. It must take two arguments: an array representing an image and a boolean array representing the mask. It may also accept **kwargs. The function must return an object with the same shape as the input spectrum.

num_cores
[int or None] The number of cores to use if running in parallel.

verbose
[int] Verbosity level to pass to joblib.

use_memmap
[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

parallel
[bool] If set to False, will force the use of a single core without using joblib.

kwargs
[dict] Passed to function.

apply_function_parallel_spectral(function, num_cores=None, verbose=0, use_memmap=True, parallel=True, **kwargs)
Apply a function in parallel along the spectral dimension. The function will be performed on data with masked values replaced with the cube’s fill value.

Parameters

function
[function] The function to apply in the spectral dimension. It must take two arguments:
an array representing a spectrum and a boolean array representing the mask. It may also accept **kwargs. The function must return an object with the same shape as the input spectrum.

**Parameters**

- **num_cores**
  - [int or None] The number of cores to use if running in parallel

- **verbose**
  - [int] Verbosity level to pass to joblib

- **use_memmap**
  - [bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

- **parallel**
  - [bool] If set to False, will force the use of a single core without using joblib.

- **kwargs**
  - [dict] Passed to function

**apply_numpy_function**

```python
apply_numpy_function(function, fill=nan, reduce=True, how='auto', projection=False, unit=None, check_endian=False, progressbar=False, includemask=False, **kwargs)
```

Apply a numpy function to the cube

**Parameters**

- **function**
  - [Numpy ufunc] A numpy ufunc to apply to the cube

- **fill**
  - [float] The fill value to use on the data

- **reduce**
  - [bool] reduce indicates whether this is a reduce-like operation, that can be accumulated one slice at a time. sum/max/min are like this. argmax/argmin/stddev are not

- **how**
  - [cube | slice | ray | auto] How to compute the moment. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

- **projection**
  - [bool] Return a Projection if the resulting array is 2D or a OneDProjection if the resulting array is 1D and the sum is over both spatial axes?

- **unit**
  - [None or astropy.units.Unit] The unit to include for the output array. For example, SpectralCube.max calls SpectralCube.apply_numpy_function(np.max, unit=self.unit), inheriting the unit from the original cube. However, for other numpy functions, e.g. numpy.argmax, the return is an index and therefore unitless.

- **check_endian**
  - [bool] A flag to check the endianness of the data before applying the function. This is only needed for optimized functions, e.g. those in the bottleneck package.

- **progressbar**
  - [bool] Show a progressbar while iterating over the slices through the cube?

- **kwargs**
  - [dict] Passed to the numpy function.
Returns

result

[Projection or Quantity or float] The result depends on the value of axis, projection, and unit. If axis is None, the return will be a scalar with or without units. If axis is an integer, the return will be a Projection if projection is set

argmax(axis=None, how='auto', **kwargs)

Return the index of the maximum data value.

The return value is arbitrary if all pixels along axis are excluded from the mask.

Ignores excluded mask elements.

Parameters

axis

[int (optional)] The axis to collapse, or None to perform a global aggregation

how

[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

argmin(axis=None, how='auto', **kwargs)

Return the index of the minimum data value.

The return value is arbitrary if all pixels along axis are excluded from the mask

Ignores excluded mask elements.

Parameters

axis

[int (optional)] The axis to collapse, or None to perform a global aggregation

how

[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

chunked(chunksize=1000)

Not Implemented.

Iterate over chunks of valid data

closest_spectral_channel(value)

Find the index of the closest spectral channel to the specified spectral coordinate.

Parameters

value

[Quantity] The value of the spectral coordinate to search for.

convolve_to(beam, convolve=<function convolve_fft>, update_function=None, **kwargs)

Convolve each channel in the cube to a specified beam
Warning: The current implementation of convolve_to creates an in-memory copy of the whole cube to store the convolved data. Issue #506 notes that this is a problem, and it is on our to-do list to fix.

Parameters

beam
[radio_beam.Beam] The beam to convolve to

convolve
[function] The astropy convolution function to use, either astropy.convolution.convolve or astropy.convolution.convolve_fft

update_function
[method] Method that is called to update an external progressbar If provided, it disables the default astropy.utils.console.ProgressBar

kwargs
[dict] Keyword arguments to pass to the convolution function

Returns

cube
[SpectralCube] A SpectralCube with a single beam
downsamp_axis

Downsample the cube by averaging over factor pixels along an axis. Crops right side if the shape is not a multiple of factor.

The WCS will be ‘downsampled’ by the specified factor as well. If the downsample factor is odd, there will be an offset in the WCS.

There is both an in-memory and a memory-mapped implementation; the default is to use the memory-mapped version. Technically, the ‘large data’ warning doesn’t apply when using the memory-mapped version, but the warning is still there anyway.

Parameters

myarr
[ndarray] The array to downsample

factor
[int] The factor to downsample by

axis
[int] The axis to downsample along

estimator
[function] defaults to mean. You can downsample by summing or something else if you want a different estimator (e.g., downsampling error: you want to sum & divide by sqrt(n))

truncate
[bool] Whether to truncate the last chunk or average over a smaller number. e.g., if you downsample [1,2,3,4] by a factor of 3, you could get either [2] or [2,4] if truncate is True or False, respectively.
**use_memmap**
[bool] Use a memory map on disk to avoid loading the whole cube into memory (several times)? If set, the warning about large cubes can be ignored (though you still have to override the warning)

**progressbar**
[bool] Include a progress bar? Only works with use_memmap=True

### find_lines

**find_lines**(velocity_offset=None, velocity_convention=None, rest_value=None, **kwargs)
Using astroquery’s splatalogue interface, search for lines within the spectral band. See astroquery.splatalogue.Splatalogue for information on keyword arguments

#### Parameters

**velocity_offset**
[u.km/u.s equivalent] An offset by which the spectral axis should be shifted before searching splatalogue. This value will be added to the velocity, so if you want to redshift a spectrum, make this value positive, and if you want to un-redshift it, make this value negative.

**velocity_convention**
[‘radio’, ‘optical’, ‘relativistic’] The doppler convention to pass to with_spectral_unit

**rest_value**
[u.GHz equivalent] The rest frequency (or wavelength or energy) to be passed to with_spectral_unit

**flattened**(slice=(), weights=None)
Return a slice of the cube giving only the valid data (i.e., removing bad values)

#### Parameters

**slice:** 3-tuple
A length-3 tuple of view (or any equivalent valid slice of a cube)

**weights:** (optional) np.ndarray
An array with the same shape (or slicing abilities/results) as the data cube

**flattened_world**(view=())
Retrieve the world coordinates corresponding to the extracted flattened version of the cube

**get_mask_array()**
Convert the mask to a boolean numpy array

**linewidth_fwhm**(how='auto')
Compute a (FWHM) linewidth map along the spectral axis.
For an explanation of the how parameter, see moment().

**linewidth_sigma**(how='auto')
Compute a (sigma) linewidth map along the spectral axis.
For an explanation of the how parameter, see moment().

**mad_std**(axis=None, how='cube', **kwargs)
Use astropy’s mad_std to computer the standard deviation
Ignores excluded mask elements.
Parameters

axis
[int (optional)] The axis to collapse, or None to perform a global aggregation

how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

mask_channels(goodchannels)
Helper function to mask out channels. This function is equivalent to adding a mask with `cube[view]` where `view` is broadcastable to the cube shape, but it accepts 1D arrays that are not normally broadcastable.

Parameters

goodchannels
[array] A 1D boolean array declaring which channels should be kept.

Returns

cube
[SpectralCube] A cube with the specified channels masked

max(axis=None, how='auto', **kwargs)
Return the maximum data value of the cube, optionally over an axis.
Ignores excluded mask elements.

Parameters

axis
[int (optional)] The axis to collapse, or None to perform a global aggregation

how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

mean(axis=None, how='cube', **kwargs)
Return the mean of the cube, optionally over an axis.
Ignores excluded mask elements.

Parameters

axis
[int (optional)] The axis to collapse, or None to perform a global aggregation

how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'
median(axis=None, iterate_rays=False, **kwargs)
Compute the median of an array, optionally along an axis.

Ignores excluded mask elements.

Parameters

axis
[int (optional)] The axis to collapse

iterate_rays
[bool] Iterate over individual rays? This mode is slower but can save RAM costs, which
may be extreme for large cubes

Returns

med
[ndarray] The median

min(axis=None, how='auto', **kwargs)
Return the minimum data value of the cube, optionally over an axis.

Ignores excluded mask elements.

Parameters

axis
[int (optional)] The axis to collapse, or None to perform a global aggregation

how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the
same result, but certain strategies are more efficient depending on data size and lay-
out. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. De-
fault='auto'

minimal_subcube(spatial_only=False)
Return the minimum enclosing subcube where the mask is valid

Parameters

spatial_only: bool
Only compute the minimal subcube in the spatial dimensions

moment(order=0, axis=0, how='auto')
Compute moments along the spectral axis.

Moments are defined as follows:

Moment 0:

\[ M_0 \int I dl \]

Moment 1:

\[ M_1 = \frac{\int I dl}{M_0} \]

Moment N:

\[ M_N = \frac{\int I(l - M_1)^N dl}{M_0} \]
**Warning:** Note that these follow the mathematical definitions of moments, and therefore the second moment will return a variance map. To get linewidth maps, you can instead use the `linewidth_fwhm()` or `linewidth_sigma()` methods.

**Parameters**

- **order**
  - `[int]` The order of the moment to take. Default=0
- **axis**
  - `[int]` The axis along which to compute the moment. Default=0
- **how**
  - `[cube | slice | ray | auto]` How to compute the moment. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**Returns**

- **map [, wcs]**
  - The moment map (numpy array) and, if wcs=True, the WCS object describing the map

**Notes**

Generally, how='cube' is fastest for small cubes that easily fit into memory. how='slice' is best for most larger datasets. how='ray' is probably only a good idea for very large cubes whose data are contiguous over the axis of the moment map.

For the first moment, the result for axis=1, 2 is the angular offset relative to the cube face. For axis=0, it is the absolute velocity/frequency of the first moment.

`moment0(axis=0, how='auto')`
Compute the zeroth moment along an axis.
See `moment()`.

`moment1(axis=0, how='auto')`
Compute the 1st moment along an axis.
For an explanation of the axis and how parameters, see `moment()`.

`moment2(axis=0, how='auto')`
Compute the 2nd moment along an axis.
For an explanation of the axis and how parameters, see `moment()`.

`percentile(q, axis=0, iterate_rays=False, **kwargs)`
Return percentiles of the data.

**Parameters**

- **q**
  - `[float]` The percentile to compute
- **axis**
  - `[int, or None]` Which axis to compute percentiles over
iterate_rays
[bool] Iterate over individual rays? This mode is slower but can save RAM costs, which may be extreme for large cubes

classmethod read(filename, format=None, hdu=None, **kwargs)
Read a spectral cube from a file.
If the file contains Stokes axes, they will automatically be dropped. If you want to read in all Stokes information, use read() instead.

Parameters

filename
[str] The file to read the cube from

format
[st] The format of the file to read. (Currently limited to ‘fits’ and ‘casa_image’)

hdu
[int or str] For FITS files, the HDU to read in (can be the ID or name of an HDU).

kwargs
[dict] If the format is ‘fits’, the kwargs are passed to open().

reproject(header, order='bilinear', use_memmap=False, filled=True)
Spatially reproject the cube into a new header. Fills the data with the cube’s fill_value to replace bad values before reprojection.
If you want to reproject a cube both spatially and spectrally, you need to use spectral_interpolate as well.

Warning: The current implementation of reproject requires that the whole cube be loaded into memory. Issue #506 notes that this is a problem, and it is on our to-do list to fix.

Parameters

header
[astropy.io.fits.Header] A header specifying a cube in valid WCS

order
[int or str, optional] The order of the interpolation (if mode is set to 'interpolation'). This can be either one of the following strings:
• ‘nearest-neighbor’
• ‘bilinear’
• ‘biquadratic’
• ‘bicubic’
or an integer. A value of 0 indicates nearest neighbor interpolation.

use_memmap
[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

filled
[bool] Fill the masked values with the cube’s fill value before reprojection? Note that
setting `filled=False` will use the raw data array, which can be a workaround that prevents loading large data into memory.

```python
sigma_clip(threshold, verbose=0, use_memmap=True, num_cores=None, **kwargs)
```

Run astropy’s sigma clipper, converting all bad values to NaN.

**Parameters**

- `threshold` ([float]) The sigma parameter in `astropy.stats.sigma_clip`, which refers to the number of sigma above which to cut.
- `verbose` ([int]) Verbosity level to pass to joblib
- `num_cores` ([int or None]) The number of cores to use when applying this function in parallel across the cube.
- `use_memmap` ([bool]) If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

```python
spatial_smooth(kernel, convolve=<function convolve>, **kwargs)
```

Smooth the image in each spatial-spatial plane of the cube.

**Parameters**

- `kernel` ([Kernel2D]) A 2D kernel from astropy
- `convolve` ([function]) The astropy convolution function to use, either `astropy.convolution.convolve` or `astropy.convolution.convolve_fft`

```python
spatial_smooth_median(ksize, update_function=None, **kwargs)
```

Smooth the image in each spatial-spatial plane of the cube using a median filter.

**Parameters**

- `ksize` ([int]) Size of the median filter (scipy.ndimage.filters.median_filter)
- `update_function` ([method]) Method that is called to update an external progress bar. If provided, it disables the default `astropy.utils.console.ProgressBar`

```python
spectral_interpolate(spectral_grid, suppress_smooth_warning=False, fill_value=None, update_function=None)
```

Resample the cube spectrally onto a specific grid.

**Parameters**
**spectral_grid**
[array] An array of the spectral positions to regrid onto

**suppress_smooth_warning**
[bool] If disabled, a warning will be raised when interpolating onto a grid that does not
nyquist sample the existing grid. Disable this if you have already appropriately smoothed
the data.

**fill_value**
[float] Value for extrapolated spectral values that lie outside of the spectral range defined
in the original data. The default is to use the nearest spectral channel in the cube.

**update_function**
[method] Method that is called to update an external progressbar. If provided, it disables
the default `astropy.utils.console.ProgressBar`

**Returns**

**cube**
[SpectralCube]

**spectral_slab(lo, hi)**
Extract a new cube between two spectral coordinates

**Parameters**

**lo, hi**
[Quantity] The lower and upper spectral coordinate for the slab range. The units should
be compatible with the units of the spectral axis. If the spectral axis is in frequency-
equivalent units and you want to select a range in velocity, or vice-versa, you should first
use `with_spectral_unit()` to convert the units of the spectral axis.

**spectral_smooth(kernel, convolve=<function convolve>, verbose=0, use_memmap=True, num_cores=None, **kwargs)**
Smooth the cube along the spectral dimension

Note that the mask is left unchanged in this operation.

**Parameters**

**kernel**
[Kernel1D] A 1D kernel from astropy

**convolve**
[function] The astropy convolution function to use, either `astropy.convolution.convolve` or
`astropy.convolution.convolve_fft`

**verbose**
[int] Verbosity level to pass to joblib

**use_memmap**
[bool] If specified, a memory mapped temporary file on disk will be written to rather than
storing the intermediate spectra in memory.

**num_cores**
[int or None] The number of cores to use if running in parallel

**kwargs**
[dict] Passed to the convolve function
**spectral_smooth_median**

Smooth the cube along the spectral dimension

**Parameters**

- **ksize**
  - [int] Size of the median filter (scipy.ndimage.filters.median_filter)

- **verbose**
  - [int] Verbosity level to pass to joblib

- **use_memmap**
  - [bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

- **num_cores**
  - [int or None] The number of cores to use if running in parallel

- **kwargs**
  - [dict] Not used at the moment.

**std**

Return the standard deviation of the cube, optionally over an axis.

**Parameters**

- **axis**
  - [int (optional)] The axis to collapse, or None to perform a global aggregation

- **how**
  - [cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Default='auto'

**Other Parameters**

- **ddof**
  - [int] 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.

**subcube**

Extract a sub-cube spatially and spectrally.

**Parameters**

- **[xyz]lo/[xyz]hi**
  - [int or Quantity or min/max] The endpoints to extract. If given as a quantity, will be interpreted as World coordinates. If given as a string or int, will be interpreted as pixel coordinates.

**subcube_from_crtfregion**

Extract a masked subcube from a CRTF region.

**Parameters**
```python
crtf_region: str
    The CRTF region(s) string to extract

allow_empty: bool
    If this is False, an exception will be raised if the region contains no overlap with the cube

subcube_from_ds9region(ds9_region, allow_empty=False)
    Extract a masked subcube from a ds9 region (only functions on celestial dimensions)

Parameters

    ds9_region: str
        The DS9 region(s) to extract

    allow_empty: bool
        If this is False, an exception will be raised if the region contains no overlap with the cube

subcube_from_mask(region_mask)
    Given a mask, return the minimal subcube that encloses the mask

Parameters

    region_mask: 'masks.MaskBase' or boolean 'numpy.ndarray'
        The mask with appropriate WCS or an ndarray with matched coordinates

subcube_from_regions(region_list, allow_empty=False)
    Extract a masked subcube from a list of regions.Region object (only functions on celestial dimensions)

Parameters

    region_list: ''regions.Region'' list
        The region(s) to extract

    allow_empty: bool, optional
        If this is False, an exception will be raised if the region contains no overlap with the cube. Default is False.

subcube_slices_from_mask(region_mask, spatial_only=False)
    Given a mask, return the slices corresponding to the minimum subcube that encloses the mask

Parameters

    region_mask: 'masks.MaskBase' or boolean 'numpy.ndarray'
        The mask with appropriate WCS or an ndarray with matched coordinates

    spatial_only: bool
        Return only slices that affect the spatial dimensions; the spectral dimension will be left unchanged

sum(axis=None, how='auto', **kwargs)
    Return the sum of the cube, optionally over an axis.
    Ignores excluded mask elements.

Parameters

    axis
        [int (optional)] The axis to collapse, or None to perform a global aggregation
```

2.3. API Documentation
how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the
same result, but certain strategies are more efficient depending on data size and lay-
out. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. De-
default='auto'

to(unit, equivalencies=())
Return the cube converted to the given unit (assuming it is equivalent). If conversion was required, this
will be a copy, otherwise it will

to_ds9(ds9id=None, newframe=False)
Send the data to ds9 (this will create a copy in memory)

Parameters

ds9id: None or string
The DS9 session ID. If ‘None’, a new one will be created. To find your ds9 session
ID, open the ds9 menu option File:XPA:Information and look for the XPA_METHOD
string, e.g. XPA_METHOD: 86ab2314:60063. You would then call this function as cube.
to_ds9(‘86ab2314:60063’)

newframe: bool
Send the cube to a new frame or to the current frame?

to_glue(name=None, glue_app=None, dataset=None, start_gui=True)
Send data to a new or existing Glue application

Parameters

name
[str or None] The name of the dataset within Glue. If None, defaults to ‘SpectralCube’. If
a dataset with the given name already exists, a new dataset with “_” appended will be
added instead.

glue_app
[GlueApplication or None] A glue application to send the data to. If this is not specified, a
new glue application will be started if one does not already exist for this cube. Otherwise,
the data will be sent to the existing glue application, self._glue_app.

dataset
[glue.core.Data or None] An existing Data object to add the cube to. This is a good way
to compare cubes with the same dimensions. Supercedes glue_app

start_gui
[bool] Start the GUI when this is run. Set to False for testing.

to_pvextractor()
Open the cube in a quick viewer written in matplotlib that allows you to create PV extractions within the
GUI

to_yt(spectral_factor=1.0, nprocs=None, **kwargs)
Convert a spectral cube to a yt object that can be further analyzed in yt.

Parameters

spectral_factor
[float, optional] Factor by which to stretch the spectral axis. If set to 1, one pixel in spectral
coordinates is equivalent to one pixel in spatial coordinates.
If using yt 3.0 or later, additional keyword arguments will be passed onto yt’s ‘‘FITSDataset‘‘ constructor. See the yt documentation (http://yt-project.org/docs/3.0/examining/loading_data.html?#fits-data) for details on options for reading FITS data.

unmasked_copy()
Return a copy of the cube with no mask (i.e., all data included)

with_beam(beam)
Attach a beam object to the SpectralCube.

Parameters

beam
[Beam] Beam object defining the resolution element of the SpectralCube.

with_fill_value(fill_value)
Create a new object with a different fill_value.

Notes
This method is fast (it does not copy any data)

with_mask(mask, inherit_mask=True, wcs_tolerance=None)
Return a new SpectralCube instance that contains a composite mask of the current SpectralCube and the new mask. Values of the mask that are True will be included (masks are analogous to numpy boolean index arrays, they are the inverse of the .mask attribute of a numpy masked array).

Parameters

mask
[MaskBase instance, or boolean numpy array] The mask to apply. If a boolean array is supplied, it will be converted into a mask, assuming that True values indicate included elements.

inherit_mask
[bool (optional, default=True)] If True, combines the provided mask with the mask currently attached to the cube

wcs_tolerance
[None or float] The tolerance of difference in WCS parameters between the cube and the mask. Defaults to self._wcs_tolerance (which itself defaults to 0.0) if unspecified

Returns

new_cube
[SpectralCube] A cube with the new mask applied.
Notes

This operation returns a view into the data, and not a copy.

with_spectral_unit(unit, velocity_convention=None, rest_value=None)

Returns a new Cube with a different Spectral Axis unit

Parameters

unit

[Unit] Any valid spectral unit: velocity, (wave)length, or frequency. Only vacuum units are supported.

velocity_convention

['relativistic', 'radio', or 'optical'] The velocity convention to use for the output velocity axis. Required if the output type is velocity. This can be either one of the above strings, or an astropy.units equivalency.

rest_value

[Quantity] A rest wavelength or frequency with appropriate units. Required if output type is velocity. The cube’s WCS should include this already if the input type is velocity, but the WCS’s rest wavelength/frequency can be overridden with this parameter.

world_spines()

Returns a list of 1D arrays, for the world coordinates along each pixel axis.

Raises error if this operation is ill-posed (e.g. rotated world coordinates, strong distortions)

This method is not currently implemented. Use world() instead.

write(filename, overwrite=False, format=None)

Write the spectral cube to a file.

Parameters

filename

[str] The path to write the file to

format

[str] The format of the file to write. (Currently limited to ‘fits’)

overwrite

[bool] If True, overwrite filename if it exists

StokesSpectralCube

class spectral_cube.StokesSpectralCube(stokes_data, mask=None, meta=None, fill_value=None)

Bases: object

A class to store a spectral cube with multiple Stokes parameters.

The individual Stokes cubes can share a common mask in addition to having component-specific masks.

Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(filename[, format, hdu])</td>
<td>Read a spectral cube from a file.</td>
</tr>
</tbody>
</table>
with_mask(mask[, inherit_mask])  
Return a new StokesSpectralCube instance that contains a composite mask of the current StokesSpectralCube and the new mask.

with_spectral_unit(unit, **kwargs)
write(filename[, overwrite, format])  
Write the spectral cube to a file.

Methods Documentation

classmethod read(filename, format=None, hdu=None, **kwargs)
Read a spectral cube from a file.

If the file contains Stokes axes, they will be read in. If you are only interested in the unpolarized emission (I), you can use read() instead.

Parameters

filename  
[str] The file to read the cube from

format  
[str] The format of the file to read. (Currently limited to ‘fits’ and ‘casa_image’)

hdu  
[int or str] For FITS files, the HDU to read in (can be the ID or name of an HDU).

Returns

cube  
[SpectralCube]

with_mask(mask, inherit_mask=True)
Return a new StokesSpectralCube instance that contains a composite mask of the current StokesSpectralCube and the new mask.

Parameters

mask  
[MaskBase instance, or boolean numpy array] The mask to apply. If a boolean array is supplied, it will be converted into a mask, assuming that True values indicate included elements.

inherit_mask  
[bool (optional, default=True)] If True, combines the provided mask with the mask currently attached to the cube

Returns

new_cube  
[StokesSpectralCube] A cube with the new mask applied.

Notes

This operation returns a view into the data, and not a copy.
with_spectral_unit(unit, **kwargs)

write(filename, overwrite=False, format=None)
Write the spectral cube to a file.

Parameters

filename
[std] The path to write the file to
format
[std] The format of the file to write. (Currently limited to `fits`)
overwrite
[bool] If True, overwrite filename if it exists

VaryingResolutionSpectralCube

class spectral_cube.VaryingResolutionSpectralCube(*args, **kwargs)
A variant of the SpectralCube class that has PSF (beam) information on a per-channel basis.

Create a SpectralCube with an associated beam table. The new VaryingResolutionSpectralCube will have a
beams attribute and a beam_threshold attribute as described below. It will perform some additional checks
when trying to perform analysis across image frames.

Three new keyword arguments are accepted:

Other Parameters

beam_table
[numpy.recarray] A table of beam major and minor axes in arcseconds and position angles,
with labels BMAJ, BMIN, BPA
beams
[list] A list of radio_beam.Beam objects
beam_threshold
[float or dict] The fractional threshold above which beams are considered different. A
dictionary may be used with entries `area`, `major`, `minor`, `pa` so that you can specify a
different fractional threshold for each of these. For example, if you want to check only that
the areas are the same, and not worry about the shape (which might be a bad idea...), you
could set beam_threshold={'area':0.01, 'major':1.5, 'minor':1.5, 'pa':5.0}

Methods Summary

**apply_function(function[, axis, weights,...])**  Apply a function to valid data along the specified
axis or to the whole cube, optionally using a weight array that is the same shape (or at least can be sliced
in the same way)

Continued on next page
Table 18 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply_function_parallel_spatial(function[, ...])</code></td>
<td>Apply a function in parallel along the spatial dimension.</td>
</tr>
<tr>
<td><code>apply_function_parallel_spectral(function[, ...])</code></td>
<td>Apply a function in parallel along the spectral dimension.</td>
</tr>
<tr>
<td><code>apply_numpy_function(function[, fill, ...])</code></td>
<td>Apply a numpy function to the cube.</td>
</tr>
<tr>
<td><code>argmax(axis, how)</code></td>
<td>Return the index of the maximum data value.</td>
</tr>
<tr>
<td><code>argmin(axis, how)</code></td>
<td>Return the index of the minimum data value.</td>
</tr>
<tr>
<td><code>average_beams(threshold[, mask, warn])</code></td>
<td>Average the beams.</td>
</tr>
<tr>
<td><code>chunked([chunksizes])</code></td>
<td>Not Implemented.</td>
</tr>
<tr>
<td><code>closest_spectral_channel(value)</code></td>
<td>Find the index of the closest spectral channel to the specified spectral coordinate.</td>
</tr>
<tr>
<td><code>convolve_to(beam[, allow_smaller, convolve, ...])</code></td>
<td>Convolve each channel in the cube to a specified beam.</td>
</tr>
<tr>
<td><code>downsample_axis(factor, axis[, estimator, ...])</code></td>
<td>Downsample the cube by averaging over <code>factor</code> pixels along an axis.</td>
</tr>
<tr>
<td><code>filled([fill_value])</code></td>
<td></td>
</tr>
<tr>
<td><code>flattened([slice, weights])</code></td>
<td>Return a slice of the cube giving only the valid data (i.e., removing bad values).</td>
</tr>
<tr>
<td><code>flattened_world([view])</code></td>
<td>Retrieve the world coordinates corresponding to the extracted flattened version of the cube.</td>
</tr>
<tr>
<td><code>get_mask_array()</code></td>
<td>Convert the mask to a boolean numpy array.</td>
</tr>
<tr>
<td><code>identify_bad_beams(threshold[, ...])</code></td>
<td>Mask out any layers in the cube that have beams that differ from the central value of the beam by more than the specified threshold.</td>
</tr>
<tr>
<td><code>jtok_factors([equivalencies])</code></td>
<td>Compute an array of multiplicative factors that will convert from Jy/beam to K.</td>
</tr>
<tr>
<td><code>linewidth_fwhm([how])</code></td>
<td>Compute a (FWHM) linewidth map along the spectral axis.</td>
</tr>
<tr>
<td><code>linewidth_sigma([how])</code></td>
<td>Compute a (sigma) linewidth map along the spectral axis.</td>
</tr>
<tr>
<td><code>mad_std([axis, how])</code></td>
<td>Use astropy's mad_std to computer the standard deviation.</td>
</tr>
<tr>
<td><code>mask_channels(goodchannels)</code></td>
<td>Helper function to mask out channels.</td>
</tr>
<tr>
<td><code>mask_out_bad_beams(threshold[, ...])</code></td>
<td>See <code>identify_bad_beams</code>.</td>
</tr>
<tr>
<td><code>max(axis, how)</code></td>
<td>Return the maximum data value of the cube, optionally over an axis.</td>
</tr>
<tr>
<td><code>mean(axis, how)</code></td>
<td>Return the mean of the cube, optionally over an axis.</td>
</tr>
<tr>
<td><code>median(axis, iterate_rays)</code></td>
<td>Compute the median of an array, optionally along an axis.</td>
</tr>
<tr>
<td><code>min(axis, how)</code></td>
<td>Return the minimum data value of the cube, optionally over an axis.</td>
</tr>
<tr>
<td><code>minimal_subcube([spatial_only])</code></td>
<td>Return the minimum enclosing subcube where the mask is valid.</td>
</tr>
<tr>
<td><code>moment([order, axis, how])</code></td>
<td>Compute moments along the spectral axis.</td>
</tr>
<tr>
<td><code>moment0([axis, how])</code></td>
<td>Compute the zeroth moment along an axis.</td>
</tr>
<tr>
<td><code>moment1([axis, how])</code></td>
<td>Compute the 1st moment along an axis.</td>
</tr>
<tr>
<td><code>moment2([axis, how])</code></td>
<td>Compute the 2nd moment along an axis.</td>
</tr>
<tr>
<td><code>percentile([q, axis, iterate_rays])</code></td>
<td>Return percentiles of the data.</td>
</tr>
<tr>
<td><code>read(filename[, format, hdu])</code></td>
<td>Read a spectral cube from a file.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reproject(header[, order, use_memmap, filled])</code></td>
<td>Spatially reproject the cube into a new header.</td>
</tr>
<tr>
<td><code>sigma_clip(threshold[, verbose, use_memmap, ...])</code></td>
<td>Run astropy’s sigma clipper, converting all bad values to NaN.</td>
</tr>
<tr>
<td><code>spatial_smooth(kernel[, convolve])</code></td>
<td>Smooth the image in each spatial-spatial plane of the cube.</td>
</tr>
<tr>
<td><code>spatial_smooth_median(ksize[, update_function])</code></td>
<td>Smooth the image in each spatial-spatial plane of the cube using a median filter.</td>
</tr>
<tr>
<td><code>spectral_interpolate(*args, **kwargs)</code></td>
<td>Resample the cube spectrally onto a specific grid</td>
</tr>
<tr>
<td><code>spectral_slab(lo, hi)</code></td>
<td>Extract a new cube between two spectral coordinates</td>
</tr>
<tr>
<td><code>spectral_smooth(*args, **kwargs)</code></td>
<td>Smooth the cube along the spectral dimension</td>
</tr>
<tr>
<td><code>spectral_smooth_median(ksize[, use_memmap, ...])</code></td>
<td>Smooth the cube along the spectral dimension</td>
</tr>
<tr>
<td><code>std([axis, how, ddof])</code></td>
<td>Return the standard deviation of the cube, optionally over an axis.</td>
</tr>
<tr>
<td><code>subcube([xlo, xhi, ylo, yhi, zlo, zhi, ...])</code></td>
<td>Extract a sub-cube spatially and spectrally.</td>
</tr>
<tr>
<td><code>subcube_from_crtfregion(crtf_region[, ...])</code></td>
<td>Extract a masked subcube from a CRTF region.</td>
</tr>
<tr>
<td><code>subcube_from_ds9region(ds9_region[, allow_empty])</code></td>
<td>Extract a masked subcube from a ds9 region (only functions on celestial dimensions)</td>
</tr>
<tr>
<td><code>subcube_from_mask(region_mask)</code></td>
<td>Given a mask, return the minimal subcube that encloses the mask</td>
</tr>
<tr>
<td><code>subcube_from_regions(region_list[, allow_empty])</code></td>
<td>Extract a masked subcube from a list of regions. Region object (only functions on celestial dimensions)</td>
</tr>
<tr>
<td><code>subcube_slices_from_mask(region_mask[, ...])</code></td>
<td>Given a mask, return the slices corresponding to the minimum subcube that encloses the mask</td>
</tr>
<tr>
<td><code>sum([axis, how])</code></td>
<td>Return the sum of the cube, optionally over an axis.</td>
</tr>
<tr>
<td><code>to(unit[, equivalencies])</code></td>
<td>Return the cube converted to the given unit (assuming it is equivalent).</td>
</tr>
<tr>
<td><code>to_ds9([ds9id, newframe])</code></td>
<td>Send the data to ds9 (this will create a copy in memory)</td>
</tr>
<tr>
<td><code>to_glue([name, glue_app, dataset, start_gui])</code></td>
<td>Send data to a new or existing Glue application</td>
</tr>
<tr>
<td><code>to_pvextractor()</code></td>
<td>Open the cube in a quick viewer written in matplotlib that allows you to create PV extractions within the GUI</td>
</tr>
<tr>
<td><code>to_yt([spectral_factor, nprocs])</code></td>
<td>Convert a spectral cube to a yt object that can be further analyzed in yt.</td>
</tr>
<tr>
<td><code>unmasked_copy()</code></td>
<td>Return a copy of the cube with no mask (i.e., all data included)</td>
</tr>
<tr>
<td><code>with_beams(beams[, goodbeams_mask])</code></td>
<td>Attach a new beams object to the VaryingResolution-SpectralCube.</td>
</tr>
<tr>
<td><code>with_fill_value(fill_value)</code></td>
<td>Create a new object with a different fill_value.</td>
</tr>
<tr>
<td><code>with_mask(mask[, inherit_mask, wcs_tolerance])</code></td>
<td>Return a new SpectralCube instance that contains a composite mask of the current SpectralCube and the new mask.</td>
</tr>
<tr>
<td><code>with_spectral_unit(unit[, ...])</code></td>
<td>Returns a new Cube with a different Spectral Axis unit</td>
</tr>
<tr>
<td><code>world_spines()</code></td>
<td>Returns a list of 1D arrays, for the world coordinates along each pixel axis.</td>
</tr>
<tr>
<td><code>write(filename[, overwrite, format])</code></td>
<td>Write the spectral cube to a file.</td>
</tr>
</tbody>
</table>
Methods Documentation

**apply_function**(*function*, *axis=None*, *weights=None*, *unit=None*, *projection=False*, *progressbar=False*, *update_function=None*, *keep_shape=False*, **kwargs)

Apply a function to valid data along the specified axis or to the whole cube, optionally using a weight array that is the same shape (or at least can be sliced in the same way).

**Parameters**

- **function**
  - [function] A function that can be applied to a numpy array. Does not need to be nan-aware.

- **axis**
  - [1, 2, 3, or None] The axis to operate along. If None, the return is scalar.

- **weights**
  - [(optional) np.ndarray] An array with the same shape (or slicing abilities/results) as the data cube.

- **unit**
  - [(optional) Unit] The unit of the output projection or value. Not all functions should return quantities with units.

- **projection**
  - [bool] Return a projection if the resulting array is 2D?

- **progressbar**
  - [bool] Show a progressbar while iterating over the slices/rays through the cube?

- **keep_shape**
  - [bool] If True, the returned object will be the same dimensionality as the cube.

- **update_function**
  - [function] An alternative tracker for the progress of applying the function to the cube data. If progressbar is True, this argument is ignored.

**Returns**

- **result**
  - [Projection or Quantity or float] The result depends on the value of axis, projection, and unit. If axis is None, the return will be a scalar with or without units. If axis is an integer, the return will be a Projection if projection is set.

**apply_function_parallel_spatial**(*function*, *num_cores=None*, *verbose=0*, *use_memmap=True*, *parallel=True*, **kwargs)

Apply a function in parallel along the spatial dimension. The function will be performed on data with masked values replaced with the cube’s fill value.

**Parameters**

- **function**
  - [function] The function to apply in the spatial dimension. It must take two arguments: an array representing an image and a boolean array representing the mask. It may also accept **kwargs. The function must return an object with the same shape as the input spectrum.

- **num_cores**
  - [int or None] The number of cores to use if running in parallel.
verbose
[int] Verbosity level to pass to joblib

use_memmap
[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

parallel
[bool] If set to False, will force the use of a single core without using joblib.

kwargs
[dict] Passed to function

apply_function_parallel_spectral(function, num_cores=None, verbose=0, use_memmap=True, parallel=True, **kwargs)
Apply a function in parallel along the spectral dimension. The function will be performed on data with masked values replaced with the cube’s fill value.

Parameters

function
[function] The function to apply in the spectral dimension. It must take two arguments: an array representing a spectrum and a boolean array representing the mask. It may also accept **kwargs. The function must return an object with the same shape as the input spectrum.

num_cores
[int or None] The number of cores to use if running in parallel

verbose
[int] Verbosity level to pass to joblib

use_memmap
[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

parallel
[bool] If set to False, will force the use of a single core without using joblib.

kwargs
[dict] Passed to function

apply_numpy_function(function, fill=nan, reduce=True, how='auto', projection=False, unit=None, check_endian=False, progressbar=False, includemask=False, **kwargs)
Apply a numpy function to the cube

Parameters

function
[Numpy ufunc] A numpy ufunc to apply to the cube

fill
[float] The fill value to use on the data

reduce
[bool] reduce indicates whether this is a reduce-like operation, that can be accumulated one slice at a time. sum/max/min are like this. argmax/argmin/stddev are not

how
[cube | slice | ray | auto] How to compute the moment. All strategies give the same result,
but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**projection**

[bool] Return a `Projection` if the resulting array is 2D or a OneDProjection if the resulting array is 1D and the sum is over both spatial axes?

**unit**

[None or `astropy.units.Unit`] The unit to include for the output array. For example, `SpectralCube.max` calls `SpectralCube.apply_numpy_function(np.max, unit=self.unit)`, inheriting the unit from the original cube. However, for other numpy functions, e.g. `numpy.argmax`, the return is an index and therefore unitless.

**check_endian**

[bool] A flag to check the endianness of the data before applying the function. This is only needed for optimized functions, e.g. those in the bottleneck package.

**progressbar**

[bool] Show a progressbar while iterating over the slices through the cube?

**kwargs**

[dict] Passed to the numpy function.

**Returns**

**result**

[`Projection` or `Quantity` or float] The result depends on the value of `axis`, `projection`, and `unit`. If `axis` is None, the return will be a scalar with or without units. If `axis` is an integer, the return will be a `Projection` if `projection` is set

**argmax**(axis=None, how='auto', **kwargs)

Return the index of the maximum data value.

The return value is arbitrary if all pixels along `axis` are excluded from the mask.

Ignores excluded mask elements.

**Parameters**

**axis**

[int (optional)] The axis to collapse, or None to perform a global aggregation

**how**

[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**argmin**(axis=None, how='auto', **kwargs)

Return the index of the minimum data value.

The return value is arbitrary if all pixels along `axis` are excluded from the mask.

Ignores excluded mask elements.

**Parameters**

**axis**

[int (optional)] The axis to collapse, or None to perform a global aggregation
how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

average_beams(threshold, mask='compute', warn=False)
Average the beams. Note that this operation only makes sense in limited contexts! Generally one would want to convolve all the beams to a common shape, but this method is meant to handle the “simple” case when all your beams are the same to within some small factor and can therefore be arithmetically averaged.

Parameters

threshold
[float] The fractional difference between beam major, minor, and pa to permit

mask
['compute', None, or boolean array] The mask to apply to the beams. Useful for excluding bad channels and edge beams.

warn
[bool] Warn if successful?

Returns

new_beam
[radio_beam.Beam] A new radio beam object that is the average of the unmasked beams

chunked(chunksize=1000)
Not Implemented.

Iterate over chunks of valid data

closest_spectral_channel(value)
Find the index of the closest spectral channel to the specified spectral coordinate.

Parameters

value
[Quantity] The value of the spectral coordinate to search for.

convolve_to(beam, allow_smaller=False, convolve=<function convolve_fft>, update_function=None)
Convolve each channel in the cube to a specified beam

Warning: The current implementation of convolve_to creates an in-memory copy of the whole cube to store the convolved data. Issue #506 notes that this is a problem, and it is on our to-do list to fix.

Warning: Note that if there is any misalignment between the cube’s spatial pixel axes and the WCS’s spatial axes and the beams are not round, the convolution kernels used here may be incorrect. Be wary in such cases!
beam
   [radio_beam.Beam] The beam to convolve to

allow_smaller
   [bool] If the specified target beam is smaller than the beam in a channel in any dimension
   and this is False, it will raise an exception.

convolve
   [function] The astropy convolution function to use, either astropy.convolution.
   convolve or astropy.convolution.convolve_fft

update_function
   [method] Method that is called to update an external progressbar If provided, it disables
   the default astropy.utils.console.ProgressBar

Returns

cube
   [SpectralCube] A SpectralCube with a single beam

downsampling_axis
   (factor, axis, estimator=<function nanmean>, truncate=False, use_memmap=True,
   progressbar=True)

   Downsample the cube by averaging over factor pixels along an axis. Crops right side if the shape
   is not a multiple of factor.

   The WCS will be ‘downsampled’ by the specified factor as well. If the downsample factor is odd, there
   will be an offset in the WCS.

   There is both an in-memory and a memory-mapped implementation; the default is to use the memory-
   mapped version. Technically, the ‘large data’ warning doesn’t apply when using the memory-mapped
   version, but the warning is still there anyway.

Parameters

myarr
   [ndarray] The array to downsample

factor
   [int] The factor to downsample by

axis
   [int] The axis to downsample along

estimator
   [function] defaults to mean. You can downsample by summing or something else if you
   want a different estimator (e.g., downsampling error: you want to sum & divide by sqrt(n))

truncate
   [bool] Whether to truncate the last chunk or average over a smaller number. e.g., if you
   downsample \([1,2,3,4]\) by a factor of 3, you could get either \([2]\) or \([2,4]\) if truncate is True
   or False, respectively.

use_memmap
   [bool] Use a memory map on disk to avoid loading the whole cube into memory (several
   times)? If set, the warning about large cubes can be ignored (though you still have to
   override the warning)

progressbar
   [bool] Include a progress bar? Only works with use_memmap=True
filled\(fill\_value=None\)

find_lines\(velocity\_offset=None\), velocity_convention=None, rest_value=None, **kwargs\)

Using astroquery’s splatalogue interface, search for lines within the spectral band. See astroquery.splatalogue.Splatalogue for information on keyword arguments

Parameters

velocity_offset
[u.km/u.s equivalent] An offset by which the spectral axis should be shifted before searching splatalogue. This value will be added to the velocity, so if you want to redshift a spectrum, make this value positive, and if you want to un-redshift it, make this value negative.

velocity_convention
[‘radio’, ‘optical’, ‘relativistic’] The doppler convention to pass to with_spectral_unit

rest_value
[u.GHz equivalent] The rest frequency (or wavelength or energy) to be passed to with_spectral_unit

flattened\(slice=(), weights=None\)

Return a slice of the cube giving only the valid data (i.e., removing bad values)

Parameters

slice: 3-tuple
A length-3 tuple of view (or any equivalent valid slice of a cube)

weights: (optional) np.ndarray
An array with the same shape (or slicing abilities/results) as the data cube

flattened_world\(view=()\)

Retrieve the world coordinates corresponding to the extracted flattened version of the cube

get_mask_array()

Convert the mask to a boolean numpy array

identify_bad_beams\(threshold\), reference_beam=None, criteria=[‘sr’, ‘major’, ‘minor’], mid_value=<function nanmedian>\)

Mask out any layers in the cube that have beams that differ from the central value of the beam by more than the specified threshold.

Parameters

threshold
[float] Fractional threshold

reference_beam
[Beam] A beam to use as the reference. If unspecified, mid_value will be used to select a middle beam

criteria

mid_value
[function] The function used to determine the ‘mid’ value to compare to. This will identify the middle-valued beam area/major/minor/pa.
Returns

**include_mask**
[np.array] A boolean array where True indicates the good beams

**jtok_factors(equivalencies=())**
Compute an array of multiplicative factors that will convert from Jy/beam to K

**linewidth_fwhm(how='auto')**
Compute a (FWHM) linewidth map along the spectral axis.
For an explanation of the how parameter, see moment().

**linewidth_sigma(how='auto')**
Compute a (sigma) linewidth map along the spectral axis.
For an explanation of the how parameter, see moment().

**mad_std(axis=None, how='cube', **kwargs)**
Use astropy’s mad_std to compute the standard deviation
Ignores excluded mask elements.

Parameters

**axis**
[int (optional)] The axis to collapse, or None to perform a global aggregation

**how**
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**mask_channels(goodchannels)**
Helper function to mask out channels. This function is equivalent to adding a mask with cube[view] where view is broadcastable to the cube shape, but it accepts 1D arrays that are not normally broadcastable. Additionally, for VaryingResolutionSpectralCube s, the beams in the bad channels will not be checked when averaging, convolving, and doing other operations that are multibeam-aware.

Parameters

**goodchannels**
[array] A 1D boolean array declaring which channels should be kept.

Returns

**cube**
[SpectralCube] A cube with the specified channels masked

**mask_out_bad_beams(threshold, reference_beam=None, criteria=['sr', 'major', 'minor'], mid_value=<function nanmedian>)**
See identify_bad_beams. This function returns a masked cube

Returns

**newcube**
[VaryingResolutionSpectralCube] The cube with bad beams masked out
**max**(*axis=None, how='auto', **kwargs*)
Return the maximum data value of the cube, optionally over an axis.

Ignores excluded mask elements.

**Parameters**

**axis**
[int (optional)] The axis to collapse, or None to perform a global aggregation

**how**
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**mean**(*axis=None, how='cube', **kwargs*)
Return the mean of the cube, optionally over an axis.

Ignores excluded mask elements.

**Parameters**

**axis**
[int (optional)] The axis to collapse, or None to perform a global aggregation

**how**
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**median**(*axis=None, iterate_rays=False, **kwargs*)
Compute the median of an array, optionally along an axis.

Ignores excluded mask elements.

**Parameters**

**axis**
[int (optional)] The axis to collapse

**iterate_rays**
[bool] Iterate over individual rays? This mode is slower but can save RAM costs, which may be extreme for large cubes

**Returns**

**med**
[ndarray] The median

**min**(*axis=None, how='auto', **kwargs*)
Return the minimum data value of the cube, optionally over an axis.

Ignores excluded mask elements.

**Parameters**
axis
[int (optional)] The axis to collapse, or None to perform a global aggregation

how
[cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

minimal_subcube(spatial_only=False)
Return the minimum enclosing subcube where the mask is valid

Parameters

spatial_only: bool
Only compute the minimal subcube in the spatial dimensions

moment(order=0, axis=0, how='auto')
Compute moments along the spectral axis.

Moments are defined as follows:

Moment 0:
\[ M_0 \int I dl \]

Moment 1:
\[ M_1 = \frac{\int I dl}{M_0} \]

Moment N:
\[ M_N = \frac{\int I(l - M_1)^N dl}{M_0} \]

Warning: Note that these follow the mathematical definitions of moments, and therefore the second moment will return a variance map. To get linewidth maps, you can instead use the linewidth_fwhm() or linewidth_sigma() methods.

Parameters

order
[int] The order of the moment to take. Default=0

axis
[int] The axis along which to compute the moment. Default=0

how
[cube | slice | ray | auto] How to compute the moment. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

Returns

map [, wcs]
The moment map (numpy array) and, if wcs=True, the WCS object describing the map
Notes

Generally, how='cube' is fastest for small cubes that easily fit into memory. how='slice' is best for most larger datasets. how='ray' is probably only a good idea for very large cubes whose data are contiguous over the axis of the moment map.

For the first moment, the result for axis=1, 2 is the angular offset relative to the cube face. For axis=0, it is the absolute velocity/frequency of the first moment.

```
moment0(axis=0, how='auto')
```

Compute the zeroth moment along an axis.

See `moment()`.

```
moment1(axis=0, how='auto')
```

Compute the 1st moment along an axis.

For an explanation of the axis and how parameters, see `moment()`.

```
moment2(axis=0, how='auto')
```

Compute the 2nd moment along an axis.

For an explanation of the axis and how parameters, see `moment()`.

```
percentile(q, axis=None, iterate_rays=False, **kwargs)
```

Return percentiles of the data.

```
q
  [float] The percentile to compute

axis
  [int, or None] Which axis to compute percentiles over

iterate_rays
  [bool] Iterate over individual rays? This mode is slower but can save RAM costs, which may be extreme for large cubes
```

classmethod read(filename, format=None, hdu=None, **kwargs)

Read a spectral cube from a file.

If the file contains Stokes axes, they will automatically be dropped. If you want to read in all Stokes information, use `read()` instead.

```
Parameters

filename
  [str] The file to read the cube from

format
  [str] The format of the file to read. (Currently limited to ‘fits’ and ‘casa_image’)

hdu
  [int or str] For FITS files, the HDU to read in (can be the ID or name of an HDU).

kwargs
  [dict] If the format is ‘fits’, the kwargs are passed to open().
```

```
reproject(header, order='bilinear', use_memmap=False, filled=True)
```

Spatially reproject the cube into a new header. Fills the data with the cube’s fill_value to replace bad values before reprojection.
If you want to reproject a cube both spatially and spectrally, you need to use `spectral_interpolate` as well.

**Warning:** The current implementation of `reproject` requires that the whole cube be loaded into memory. Issue #506 notes that this is a problem, and it is on our to-do list to fix.

**Parameters**

header

[astropy.io.fits.Header] A header specifying a cube in valid WCS order

[<class 'int'> or <class 'str'>, optional] The order of the interpolation (if mode is set to ‘interpolation’). This can be either one of the following strings:

- ‘nearest-neighbor’
- ‘bilinear’
- ‘biquadratic’
- ‘bicubic’

or an integer. A value of 0 indicates nearest neighbor interpolation.

use_memmap

[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

filled

[bool] Fill the masked values with the cube’s fill value before reprojection? Note that setting filled=False will use the raw data array, which can be a workaround that prevents loading large data into memory.

**sigma_clip**(threshold, verbose=0, use_memmap=True, num_cores=None, **kwargs)

Run astropy’s sigma clipper, converting all bad values to NaN.

**Parameters**

threshold

[float] The sigma parameter in `astropy.stats.sigma_clip`, which refers to the number of sigma above which to cut.

verbose

[int] Verbosity level to pass to joblib

num_cores

[<class 'int'> or <None>] The number of cores to use when applying this function in parallel across the cube.

use_memmap

[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

**spatial_smooth**(kernel, convolve=<function convolve>, **kwargs)

Smooth the image in each spatial-spatial plane of the cube.
Parameters

**kernel**

[Kernel2D] A 2D kernel from astropy

**convolve**

[function] The astropy convolution function to use, either `astropy.convolution.convolve` or `astropy.convolution.convolve_fft`

**kwargs**

[dict] Passed to the convolve function

**spatial_smooth_median**(*ksize, update_function=None, **kwargs*)

Smooth the image in each spatial-spatial plane of the cube using a median filter.

Parameters

**ksize**

[int] Size of the median filter (scipy.ndimage.filters.median_filter)

**update_function**

[method] Method that is called to update an external progressbar. If provided, it disables the default `astropy.utils.console.ProgressBar`

**kwargs**

[dict] Passed to the convolve function

**spectral_interpolate**(*args, **kwargs*)

Resample the cube spectrally onto a specific grid

Parameters

**spectral_grid**

[array] An array of the spectral positions to regrid onto

**suppress_smooth_warning**

[bool] If disabled, a warning will be raised when interpolating onto a grid that does not nyquist sample the existing grid. Disable this if you have already appropriately smoothed the data.

**fill_value**

[float] Value for extrapolated spectral values that lie outside of the spectral range defined in the original data. The default is to use the nearest spectral channel in the cube.

**update_function**

[method] Method that is called to update an external progressbar. If provided, it disables the default `astropy.utils.console.ProgressBar`

Returns

**cube**

[SpectralCube]

**spectral_slab**(*lo, hi*)

Extract a new cube between two spectral coordinates

Parameters
lo, hi

[Quantity] The lower and upper spectral coordinate for the slab range. The units should be compatible with the units of the spectral axis. If the spectral axis is in frequency-equivalent units and you want to select a range in velocity, or vice-versa, you should first use with_spectral_unit() to convert the units of the spectral axis.

**spectral_smooth(**args, **kwargs)**

Smooth the cube along the spectral dimension

Note that the mask is left unchanged in this operation.

**Parameters**

-kernel

[Kernel1D] A 1D kernel from astropy

-convolve

[function] The astropy convolution function to use, either astropy.convolution.convolve or astropy.convolution.convolve_fft

-verbose

[int] Verbosity level to pass to joblib

-use_memmap

[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

-num_cores

[int or None] The number of cores to use if running in parallel

-kwarg[1]

[dict] Passed to the convolve function

**spectral_smooth_median**(ksize, use_memmap=True, verbose=0, num_cores=None, **kwargs)

Smooth the cube along the spectral dimension

**Parameters**

-ksize

[int] Size of the median filter (scipy.ndimage.filters.median_filter)

-verbose

[int] Verbosity level to pass to joblib

-use_memmap

[bool] If specified, a memory mapped temporary file on disk will be written to rather than storing the intermediate spectra in memory.

-num_cores

[int or None] The number of cores to use if running in parallel

-kwarg[1]

[dict] Not used at the moment.

**std**(axis=None, how='cube', ddof=0, **kwargs)

Return the standard deviation of the cube, optionally over an axis.

**Parameters**

-axis

[None, how] cube, ddof=0, **kwargs)
axis
    [int (optional)] The axis to collapse, or None to perform a global aggregation

how
    [cube | slice | ray | auto] How to compute the aggregation. All strategies give the
    same result, but certain strategies are more efficient depending on data size and lay-
    out. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. De-
    fault='auto'

Other Parameters

ddof
    [int] 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.

Ignores excluded mask elements.

\textbf{subcube}(xlo='min', xhi='max', ylo='min', yhi='max', zlo='min', zhi='max', rest_value=None)
Extract a sub-cube spatially and spectrally.

 Parameters

[xyz]lo/[xyz]hi
    [int or Quantity or min/max] The endpoints to extract. If given as a quantity, will be
    interpreted as World coordinates. If given as a string or int, will be interpreted as pixel
    coordinates.

\textbf{subcube_from_crtfregion}(crtf_region, allow_empty=False)
Extract a masked subcube from a CRTF region.

 Parameters

\textbf{crtf_region}: str
    The CRTF region(s) string to extract

\textbf{allow_empty}: bool
    If this is False, an exception will be raised if the region contains no overlap with the cube

\textbf{subcube_from_ds9region}(ds9_region, allow_empty=False)
Extract a masked subcube from a ds9 region (only functions on celestial dimensions)

 Parameters

\textbf{ds9_region}: str
    The DS9 region(s) to extract

\textbf{allow_empty}: bool
    If this is False, an exception will be raised if the region contains no overlap with the cube

\textbf{subcube_from_mask}(region_mask)
Given a mask, return the minimal subcube that encloses the mask

 Parameters

\textbf{region_mask}: ‘masks.MaskBase‘ or boolean ‘numpy.ndarray‘
    The mask with appropraite WCS or an ndarray with matched coordinates
**subcube_from_regions** *(region_list, allow_empty=False)*

Extract a masked subcube from a list of regions.Region object (only functions on celestial dimensions)

**Parameters**

- **region_list**: `regions.Region` list
  - The region(s) to extract
- **allow_empty**: bool, optional
  - If this is False, an exception will be raised if the region contains no overlap with the cube.
  - Default is False.

**subcube_slices_from_mask** *(region_mask, spatial_only=False)*

Given a mask, return the slices corresponding to the minimum subcube that encloses the mask

**Parameters**

- **region_mask**: `masks.MaskBase` or boolean `numpy.ndarray`
  - The mask with appropriate WCS or an ndarray with matched coordinates
- **spatial_only**: bool
  - Return only slices that affect the spatial dimensions; the spectral dimension will be left unchanged

**sum** *(axis=None, how='auto', **kwargs)*

Return the sum of the cube, optionally over an axis.

- **axis**
  - [int (optional)] The axis to collapse, or None to perform a global aggregation
- **how**
  - [cube | slice | ray | auto] How to compute the aggregation. All strategies give the same result, but certain strategies are more efficient depending on data size and layout. Cube/slice/ray iterate over decreasing subsets of the data, to conserve memory. Default='auto'

**to** *(unit, equivalencies=())*

Return the cube converted to the given unit (assuming it is equivalent). If conversion was required, this will be a copy, otherwise it will

**to_ds9** *(ds9id=None, newframe=False)*

Send the data to ds9 (this will create a copy in memory)

**Parameters**

- **ds9id**: None or string
  - The DS9 session ID. If ‘None’, a new one will be created. To find your ds9 session ID, open the ds9 menu option File:XPA:Information and look for the XPA_METHOD string, e.g. XPA_METHOD: 86ab2314:60063. You would then call this function as cube.to_ds9('86ab2314:60063')
- **newframe**: bool
  - Send the cube to a new frame or to the current frame?
to_glue(name=None, glue_app=None, dataset=None, start_gui=True)

Send data to a new or existing Glue application

Parameters

name
[ str or None] The name of the dataset within Glue. If None, defaults to ‘SpectralCube’. If a dataset with the given name already exists, a new dataset with ‘_’ appended will be added instead.

glue_app
[GlueApplication or None] A glue application to send the data to. If this is not specified, a new glue application will be started if one does not already exist for this cube. Otherwise, the data will be sent to the existing glue application, self._glue_app.

dataset
[glue.core.Data or None] An existing Data object to add the cube to. This is a good way to compare cubes with the same dimensions. Supercedes glue_app

start_gui
[bool] Start the GUI when this is run. Set to False for testing.

to_pvextractor()

Open the cube in a quick viewer written in matplotlib that allows you to create PV extractions within the GUI

to_yt(spectral_factor=1.0, nprocs=None, **kwargs)

Convert a spectral cube to a yt object that can be further analyzed in yt.

Parameters

spectral_factor
[float, optional] Factor by which to stretch the spectral axis. If set to 1, one pixel in spectral coordinates is equivalent to one pixel in spatial coordinates.

If using yt 3.0 or later, additional keyword arguments will be passed onto yt’s “FITSDataset“ constructor. See the yt documentation (http://yt-project.org/docs/3.0/examining/loading_data.html?#fits-data) for details on options for reading FITS data.

unmasked_copy()

Return a copy of the cube with no mask (i.e., all data included)

with_beams(beams, goodbeams_mask=None)

Attach a new beams object to the VaryingResolutionSpectralCube.

Parameters

beams

with_fill_value(fill_value)

Create a new object with a different fill_value.
Notes

This method is fast (it does not copy any data)

with_mask(mask, inherit_mask=True, wcs_tolerance=None)
Return a new SpectralCube instance that contains a composite mask of the current SpectralCube and the
new mask. Values of the mask that are True will be included (masks are analogous to numpy boolean
index arrays, they are the inverse of the .mask attribute of a numpy masked array).

Parameters

mask
[MaskBase instance, or boolean numpy array] The mask to apply. If a boolean array is
supplied, it will be converted into a mask, assuming that True values indicate included
elements.

inherit_mask
[bool (optional, default=True)] If True, combines the provided mask with the mask cur-
rently attached to the cube

wcs_tolerance
[None or float] The tolerance of difference in WCS parameters between the cube and the
mask. Defaults to self._wcs_tolerance (which itself defaults to 0.0) if unspecified

Returns

new_cube
[SpectralCube] A cube with the new mask applied.

Notes

This operation returns a view into the data, and not a copy.

with_spectral_unit(unit, velocity_convention=None, rest_value=None)
Returns a new Cube with a different Spectral Axis unit

Parameters

unit
[Unit] Any valid spectral unit: velocity, (wave)length, or frequency. Only vacuum units
are supported.

velocity_convention
[‘relativistic’, ‘radio’, or ‘optical’] The velocity convention to use for the output velocity
axis. Required if the output type is velocity. This can be either one of the above strings, or
an astropy.units equivalency.

rest_value
[Quantity] A rest wavelength or frequency with appropriate units. Required if output
type is velocity. The cube’s WCS should include this already if the input type is velocity,
but the WCS’s rest wavelength/frequency can be overridden with this parameter.

world_spines()
Returns a list of 1D arrays, for the world coordinates along each pixel axis.

Raises error if this operation is ill-posed (e.g. rotated world coordinates, strong distortions)

This method is not currently implemented. Use world() instead.
write(filename, overwrite=False, format=None)
Write the spectral cube to a file.

Parameters

filename
[str] The path to write the file to

format
[str] The format of the file to write. (Currently limited to ‘fits’)

overwrite
[bool] If True, overwrite filename if it exists

2.3.2 spectral_cube.ytcube Module

Classes

ytCube(cube, dataset[, spectral_factor]) Light wrapper of a yt object with ability to translate yt<->wcs coordinates

class spectral_cube.ytcube.ytCube(cube, dataset, spectral_factor=1.0)
Bases: object
Light wrapper of a yt object with ability to translate yt<->wcs coordinates

Methods Summary

auto_transfer_function(cmap_range[, log, ...])
quick_isocontour([level, title, ...])
quick_render_movie(outdir[, size, nframes, ...])
world2yt(world_coord[, first_index])
yt2world(yt_coord[, first_index])

Methods Documentation

auto_transfer_function(cmap_range, log=False, colormap='doom', **kwargs)

quick_isocontour(level='3 sigma', title='', description='', color_map='hot', color_log=False, export_to='sketchfab', filename=None, **kwargs)
Export isocontours to sketchfab
Requires that you have an account on https://sketchfab.com and are logged in

Parameters
level: str or float
The level of the isocontours to create. Can be specified as n-sigma with strings like ‘3.3 sigma’ or ‘2 sigma’ (there must be a space between the number and the word)

title: str
A title for the uploaded figure
description: str
A short description for the uploaded figure
color_map: str
Any valid colormap. See yt.show_colormaps
color_log: bool
Whether the colormap should be log scaled. With the default parameters, this has no effect.
export_to: ‘sketchfab’, ‘obj’, ‘ply’
You can export to sketchfab, to a .obj file (and accompanying .mtl file), or a .ply file. The latter two require filename specification
filename: None or str
Optional - prefix for output filenames if export_to is ‘obj’, or the full filename when export_to is ‘ply’. Ignored for ‘sketchfab’
kwarg: dict
Keyword arguments are passed to the appropriate yt function

Returns
The result of the ‘yt.surface.export_sketchfab’ function

quick_render_movie(outdir, size=256, nframes=30, camera_angle=(0, 0, 1), north_vector=(0, 0, 1), rot_vector=(1, 0, 0), colormap='doom', cmap_range='auto', transfer_function='auto', start_index=0, image_prefix=", output_filename='out.mp4', log_scale=False, rescale=True)
Create a movie rotating the cube 360 degrees from PP -> PV -> PP -> PV -> PP

Parameters

outdir: str
The output directory in which the individual image frames and the resulting output mp4 file should be stored

size: int
The size of the individual output frame in pixels (i.e., size=256 will result in a 256x256 image)
nframes: int
The number of frames in the resulting movie

camera_angle: 3-tuple
The initial angle of the camera

north_vector: 3-tuple
The vector of ‘north’ in the data cube. Default is coincident with the spectral axis

rot_vector: 3-tuple
The vector around which the camera will be rotated
colormap: str
   A valid colormap. See yt.show_colormaps

transfer_function: ‘auto’ or ‘yt.visualization.volume_rendering.TransferFunction’
   Either ‘auto’ to use the colormap specified, or a valid TransferFunction instance

log_scale: bool
   Should the colormap be log scaled?

rescale: bool
   If True, the images will be rescaled to have a common 95th percentile brightness, which
   can help reduce flickering from having a single bright pixel in some projections

start_index
   [int] The number of the first image to save

image_prefix
   [str] A string to prepend to the image name for each image that is output

output_filename
   [str] The movie file name to output. The suffix may affect the file type created. Defaults
to ‘out.mp4’. Will be placed in outdir

world2yt(world_coord, first_index=0)
   Convert a position in world coordinates to the coordinates used by a yt dataset that has been generated
   using the to_yt method.

   Parameters

   world_coord: ‘astropy.wcs.WCS.wcs_world2pix’-valid input
      The world coordinates

   first_index: 0 or 1
      The first index of the data. In python and yt, this should be zero, but for the FITS coordi-
      nates, use 1

yt2world(yt_coord, first_index=0)
   Convert a position in yt’s coordinates to world coordinates from a yt dataset that has been generated using
   the to_yt method.

   Parameters

   world_coord: ‘astropy.wcs.WCS.wcs_pix2world’-valid input
      The yt pixel coordinates to convert back to world coordinates

   first_index: 0 or 1
      The first index of the data. In python and yt, this should be zero, but for the FITS coordi-
      nates, use 1

2.3.3 spectral_cube.io.casa_masks Module

Functions

make_casa_mask(SpecCube, outname[, . . . ])
   Outputs the mask attached to the SpectralCube object as a CASA image, or optionally appends the mask to a
   preexisting CASA image.
**make_casa_mask**

```python
spectral_cube.io.casa_masks.make_casa_mask(SpecCube, outname, append_to_image=True,
img=None, add_stokes=True, stokes_posn=None,
overwrite=False)
```

Outputs the mask attached to the SpectralCube object as a CASA image, or optionally appends the mask to a preexisting CASA image.

**Parameters**

- **SpecCube**

- **outname**
  [str] Name of the outputted mask file.

- **append_to_image**
  [bool, optional] Appends the mask to a given image.

- **img**
  [str, optional] Image to be appended to. Must be specified if append_to_image is enabled.

- **add_stokes**
  [bool, optional]
  Adds a Stokes axis onto the wcs from SpecCube.

- **stokes_posn**
  [int, optional] Sets the position of the new Stokes axis. Defaults to the last axis.

- **overwrite**
  [bool, optional] Overwrite the image and mask files if they exist?

### 2.3.4 spectral_cube.lower_dimensional_structures Module

**Classes**

- LowerDimensionalObject
  Generic class for 1D and 2D objects.

- Projection

- Slice

- OneDSpectrum

**LowerDimensionalObject**

```python
class spectral_cube.lower_dimensional_structures.LowerDimensionalObject
Bases: astropy.units.quantity.Quantity, spectral_cube.base_class.BaseNDClass, spectral_cube.base_class.HeaderMixinClass
```

Generic class for 1D and 2D objects.

**Methods Summary**

- **shrink_mask()**
  Copy of the numpy masked_array shrink_mask method.
Table 23 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to(unit[, equivalencies, freq])</code></td>
<td>Return a new <code>Projection</code> of the same class with the specified unit.</td>
</tr>
<tr>
<td><code>write(filename[, format, overwrite])</code></td>
<td>Write the lower dimensional object to a file.</td>
</tr>
</tbody>
</table>

**Methods Documentation**

**shrink_mask()**
Copy of the numpy masked_array shrink_mask method. This is essentially a hack needed for matplotlib to show images.

```python
shrink_mask() -> copy
```

**to(unit, equivalencies=[], freq=None)**
Return a new `Projection` of the same class with the specified unit.

See `astropy.units.Quantity.to` for further details.

```python
to(unit, equivalencies=[], freq=None) -> copy
```

**write(filename, format=None, overwrite=False)**
Write the lower dimensional object to a file.

Parameters

- **filename** [str] The path to write the file to
- **format** [str] The kind of file to write. (Currently limited to ‘fits’)
- **overwrite** [bool] If True, overwrite `filename` if it exists

**Projection**

class spectral_cube.lower_dimensional_structures.Projection
Bases: spectral_cube.lower_dimensional_structures.LowerDimensionalObject, spectral_cube.base_class.SpatialCoordMixinClass, spectral_cube.base_class.MaskableArrayMixinClass, spectral_cube.base_class.BeamMixinClass

**Methods Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convolve_to(beam[, convolve])</td>
<td>Convolve the image to a specified beam.</td>
</tr>
<tr>
<td>from_hdu(hdu)</td>
<td>Return a projection from a FITS HDU.</td>
</tr>
<tr>
<td>quicklook([filename, use_aplpy, aplpy_kwargs])</td>
<td>Use APLPy to make a quick-look image of the projection. This will make the FITSFigure attribute available.</td>
</tr>
<tr>
<td>reproject(header[, order])</td>
<td>Reproject the image into a new header.</td>
</tr>
<tr>
<td>subimage([xlo, xhi, ylo, yhi])</td>
<td>Extract a region spatially.</td>
</tr>
<tr>
<td>to(unit[, equivalencies, freq])</td>
<td>Return a new <code>Projection</code> of the same class with the specified unit.</td>
</tr>
<tr>
<td>with_beam(beam)</td>
<td>Attach a new beam object to the <code>Projection</code>.</td>
</tr>
<tr>
<td>with_fill_value(fill_value)</td>
<td>Create a new <code>Projection</code> or <code>Slice</code> with a different fill_value.</td>
</tr>
</tbody>
</table>
Methods Documentation

**convolve_to**(beam, convolve=<function convolve_fft>)

Convolves the image to a specified beam.

**Parameters**

- **beam**
  
  [radio_beam.Beam] The beam to convolve to

- **convolve**
  
  [function] The astropy convolution function to use, either astropy.convolution.convolve or astropy.convolution.convolve_fft

**Returns**

- **proj**
  
  [Projection] A Projection convolved to the given beam object.

**static from_hdu**(hdu)

Return a projection from a FITS HDU.

**quicklook**(filename=None, use_aplpy=True, aplpy_kwargs={})

Use APL.py to make a quick-look image of the projection. This will make the FITSFigure attribute available.

If there are unmatched celestial axes, this will instead show an image without axis labels.

**Parameters**

- **filename**
  
  [str or None] Optional - the filename to save the quicklook to.

**reproject**(header, order='bilinear')

Reproject the image into a new header.

**Parameters**

- **header**
  
  [astropy.io.fits.Header] A header specifying a cube in valid WCS

- **order**
  
  [int or str, optional] The order of the interpolation (if mode is set to 'interpolation'). This can be either one of the following strings:

  - 'nearest-neighbor'
  - 'bilinear'
  - 'biquadratic'
  - 'bicubic'

  or an integer. A value of 0 indicates nearest neighbor interpolation.

**subimage**(xlo='min', xhi='max', ylo='min', yhi='max')

Extract a region spatially.

**Parameters**
[xy]lo/[xy]hi
[int or astropy.units.Quantity or min/max] The endpoints to extract. If given as a quantity, will be interpreted as World coordinates. If given as a string or int, will be interpreted as pixel coordinates.

to(unit, equivalencies=[], freq=None)
Return a new Projection of the same class with the specified unit.

See astropy.units.Quantity.to for further details.

with_beam(beam)
Attach a new beam object to the Projection.

Parameters

beam

with_fill_value(fill_value)
Create a new Projection or Slice with a different fill_value.

Slice
class spectral_cube.lower_dimensional_structures.Slice
Bases: spectral_cube.lower_dimensional_structures.Projection

OneDSpectrum
class spectral_cube.lower_dimensional_structures.OneDSpectrum
Bases: spectral_cube.lower_dimensional_structures.BaseOneDSpectrum, spectral_cube.base_class.BeamMixinClass

Methods Summary

with_beam(beam) Attach a new beam object to the OneDSpectrum.

Methods Documentation

with_beam(beam) Attach a new beam object to the OneDSpectrum.

Parameters

beam
S
spectral_cube, 37
spectral_cube.io.casa_masks, 204
spectral_cube.lower_dimensional_structures, 205
spectral_cube.ytcube, 202
INDEX

A
all() (spectral_cube.OneDSpectrum method), 60
all() (spectral_cube.Projection method), 98
all() (spectral_cube.Slice method), 135
any() (spectral_cube.BooleanArrayMask method), 39
any() (spectral_cube.CompositeMask method), 41
any() (spectral_cube.FunctionMask method), 43
any() (spectral_cube.InvertedMask method), 44
any() (spectral_cube.LazyComparisonMask method), 46
any() (spectral_cube.LazyMask method), 48
any() (spectral_cube.MaskBase method), 49
any() (spectral_cube.OneDSpectrum method), 61
any() (spectral_cube.Projection method), 98
any() (spectral_cube.Slice method), 135
apply_function() (spectral_cube.SpectralCube method), 164
apply_function() (spectral_cube.VaryingResolutionSpectralCube method), 185
apply_function_parallel_spatial() (spectral_cube.SpectralCube method), 165
apply_function_parallel_spatial() (spectral_cube.VaryingResolutionSpectralCube method), 185
apply_function_parallel_spectral() (spectral_cube.SpectralCube method), 165
apply_function_parallel_spectral() (spectral_cube.VaryingResolutionSpectralCube method), 186
apply_numpy_function() (spectral_cube.SpectralCube method), 166
apply_numpy_function() (spectral_cube.VaryingResolutionSpectralCube method), 186
argmax() (spectral_cube.OneDSpectrum method), 61
argmax() (spectral_cube.Projection method), 98
argmax() (spectral_cube.Slice method), 135
argmax() (spectral_cube.SpectralCube method), 167
argmax() (spectral_cube.VaryingResolutionSpectralCube method), 187
argpartition() (spectral_cube.OneDSpectrum method), 61
argpartition() (spectral_cube.Projection method), 98
argpartition() (spectral_cube.Slice method), 136
argpartition() (spectral_cube.SpectralCube method), 167
argpartition() (spectral_cube.VaryingResolutionSpectralCube method), 187
argsort() (spectral_cube.OneDSpectrum method), 136
argsort() (spectral_cube.Projection method), 99
argsort() (spectral_cube.Slice method), 136
argsort() (spectral_cube.SpectralCube method), 167
argsort() (spectral_cube.VaryingResolutionSpectralCube method), 187
argpartition() (spectral_cube.Slice method), 136
argpartition() (spectral_cube.SpectralCube method), 167
argpartition() (spectral_cube.VaryingResolutionSpectralCube method), 187
argpartition() (spectral_cube.VaryingResolutionSpectralCube method), 187
argmin() (spectral_cube.Slice method), 136
argmin() (spectral_cube.SpectralCube method), 167
argmin() (spectral_cube.VaryingResolutionSpectralCube method), 187
auto_transfer_function() (spectral_cube.ytcube.ytCube method), 202
average_beams() (spectral_cube.SpectralCube method), 188

B
base (spectral_cube.OneDSpectrum attribute), 53
base (spectral_cube.Projection attribute), 90
base (spectral_cube.Slice attribute), 128
BooleanArrayMask (class in spectral_cube), 39
byteswap() (spectral_cube.OneDSpectrum method), 63
byteswap() (spectral_cube.Projection method), 100
byteswap() (spectral_cube.Slice method), 137

C
choose() (spectral_cube.OneDSpectrum attribute), 63
choose() (spectral_cube.Projection attribute), 100
choose() (spectral_cube.Slice attribute), 138
chunked() (spectral_cube.SpectralCube method), 167
chunked() (spectral_cube.VaryingResolutionSpectralCube method), 188
clip() (spectral_cube.OneDSpectrum method), 63
clip() (spectral_cube.Projection method), 101
clip() (spectral_cube.Slice method), 138
closest_spectral_channel() (spectral_cube.SpectralCube method), 167
closest_spectral_channel() (spectral_cube.VaryingResolutionSpectralCube method), 188
CompositeMask (class in spectral_cube), 40
compress() (spectral_cube.OneDSpectrum method), 64
compress() (spectral_cube.Projection method), 101
compress() (spectral_cube.Slice method), 138
conjugate() (spectral_cube.OneDSpectrum method), 64
conjugate() (spectral_cube.Projection method), 101
conjugate() (spectral_cube.Slice method), 138
conjugate_to() (spectral_cube.Slice method), 139
convolve_to() (spectral_cube.lower_dimensional_structures.Projection method), 207
convolve_to() (spectral_cube.Projection method), 101
convolve_to() (spectral_cube.Slice method), 139
convolve_to() (spectral_cube.SpectralCube method), 167
convolve_to() (spectral_cube.VaryingResolutionSpectralCube method), 188
copy() (spectral_cube.OneDSpectrum method), 64
copy() (spectral_cube.Projection method), 102
copy() (spectral_cube.Slice method), 139
ctypes (spectral_cube.OneDSpectrum attribute), 54
ctypes (spectral_cube.Projection attribute), 91
ctypes (spectral_cube.Slice attribute), 128
cumprod() (spectral_cube.OneDSpectrum method), 65
cumprod() (spectral_cube.Projection method), 102
cumprod() (spectral_cube.Slice method), 140
cumsum() (spectral_cube.OneDSpectrum method), 65
cumsum() (spectral_cube.Projection method), 102
cumsum() (spectral_cube.Slice method), 140
diagonal() (spectral_cube.Slice method), 140
diff() (spectral_cube.OneDSpectrum method), 66
diff() (spectral_cube.Projection method), 103
diff() (spectral_cube.Slice method), 141
dot() (spectral_cube.OneDSpectrum method), 66
dot() (spectral_cube.Projection method), 103
dot() (spectral_cube.Slice method), 141
downsample_axis() (spectral_cube.SpectralCube method), 168
downsample_axis() (spectral_cube.VaryingResolutionSpectralCube method), 189
dtype (spectral_cube.OneDSpectrum attribute), 55
dtype (spectral_cube.Slice attribute), 92
dtype (spectral_cube.VaryingResolutionSpectralCube attribute), 130
dump() (spectral_cube.OneDSpectrum method), 66
dump() (spectral_cube.Projection method), 104
dump() (spectral_cube.Slice method), 141
dumps() (spectral_cube.OneDSpectrum method), 66
dumps() (spectral_cube.Projection method), 104
dumps() (spectral_cube.Slice method), 141

E
ediff1d() (spectral_cube.OneDSpectrum method), 66
ediff1d() (spectral_cube.Projection method), 104
ediff1d() (spectral_cube.Slice method), 141
exclude() (spectral_cube.BooleanArrayMask method), 39
exclude() (spectral_cube.CompositeMask method), 41
exclude() (spectral_cube.FunctionMask method), 43
exclude() (spectral_cube.InvertedMask method), 44
exclude() (spectral_cube.LazyComparisonMask method), 46
exclude() (spectral_cube.LazyMask method), 48
exclude() (spectral_cube.MaskBase method), 49

F
fill() (spectral_cube.OneDSpectrum method), 66
fill() (spectral_cube.Projection method), 104
fill() (spectral_cube.Slice method), 141
filled() (spectral_cube.OneDSpectrum method), 67
filled() (spectral_cube.Projection method), 104
filled() (spectral_cube.Slice method), 142
filled() (spectral_cube.SpectralCube method), 169
filled() (spectral_cube.VaryingResolutionSpectralCube method), 189
find_lines() (spectral_cube.SpectralCube method), 169
find_lines() (spectral_cube.VaryingResolutionSpectralCube method), 190
flags (spectral_cube.OneDSpectrum attribute), 56
flags (spectral_cube.Projection attribute), 93
flags (spectral_cube.Slice attribute), 130
flatten() (spectral_cube.OneDSpectrum method), 67
real (spectral_cube.Projection attribute), 95
real (spectral_cube.Slice attribute), 133
repeat() (spectral_cube.OneDSpectrum method), 74
repeat() (spectral_cube.Projection method), 111
repeat() (spectral_cube.Slice method), 149
reproject() (spectral_cube.lower_dimensional_structures.Projection method), 207
reproject() (spectral_cube.Projection method), 111
reproject() (spectral_cube.Slice method), 149
reproject() (spectral_cube.SpectralCube method), 173
reproject() (spectral_cube.VaryingResolutionSpectralCube method), 194
reshape() (spectral_cube.OneDSpectrum method), 74
reshape() (spectral_cube.Slice method), 149
reshape() (spectral_cube.SpectralCube method), 173
resize() (spectral_cube.OneDSpectrum method), 74
resize() (spectral_cube.Projection method), 112
resize() (spectral_cube.Slice method), 149
round() (spectral_cube.Slice method), 125
round() (spectral_cube.Slice method), 153
round() (spectral_cube.Slice method), 151
searchsorted() (spectral_cube.Slice method), 76
searchsorted() (spectral_cube.Slice method), 113
setfield() (spectral_cube.Slice method), 151
setfield() (spectral_cube.Slice method), 151
setfield() (spectral_cube.Slice method), 151
setflags() (spectral_cube.Slice method), 114
setflags() (spectral_cube.Slice method), 152
shape (numpy.core._internal.Slice._ctypes attribute), 54
shape (numpy.core._internal.Slice._ctypes attribute), 91
shape (numpy.core._internal.Slice._ctypes attribute), 129
shape (spectral_cube.Slice method), 58
shape (spectral_cube.Projection method), 96
shape (spectral_cube.Slice method), 133
shape_as() (spectral_cube.Slice method), 55
shape_as() (spectral_cube.Slice method), 92
shape_as() (spectral_cube.Slice method), 129
shrink_mask() (spectral_cube.lower_dimensional_structures.LowerDimensionalObject method), 206
shrink_mask() (spectral_cube.Slice method), 78
shrink_mask() (spectral_cube.Slice method), 116
shrink_mask() (spectral_cube.Slice method), 153
sigma_clip() (spectral_cube.SpectralCube method), 174
sigma_clip() (spectral_cube.VaryingResolutionSpectralCube method), 195
size (spectral_cube.OneDSpectrum attribute), 174
size (spectral_cube.Slice attribute), 134
Slice (class in spectral_cube), 125
Slice (class in spectral_cube.lower_dimensional_structures), 208
sort() (spectral_cube.Slice method), 78
sort() (spectral_cube.Slice method), 116
sort() (spectral_cube.Slice method), 153
spatial_smooth() (spectral_cube.SpectralCube method), 174
spatial_smooth() (spectral_cube.VaryingResolutionSpectralCube method), 195
spatial_smooth_median() (spectral_cube.SpectralCube method), 174
spatial_smooth_median() (spectral_cube.VaryingResolutionSpectralCube method), 196
spectral_cube (module), 37
spectral_cube.io.casa_masks (module), 204
spectral_cube.lower_dimensional_structures (module), 205
spectral_cube.ytcube (module), 202
spectral_interpolate() (spectral_cube.Slice method), 79
spectral_interpolate() (spectral_cube.SpectralCube method), 174
spectral_interpolate() (spectral_cube.VaryingResolutionSpectralCube method), 196
spectral_slab() (spectral_cube.Slice method), 175
spectral_slab() (spectral_cube.VaryingResolutionSpectralCube method), 196
spectral_smooth() (spectral_cube.Slice method), 79
spectral_smooth() (spectral_cube.SpectralCube method), 175
spectral_smooth() (spectral_cube.VaryingResolutionSpectralCube method), 197
spectral_smooth_median() (spectral_cube.Slice method), 175
spectral_smooth_median() (spectral_cube.VaryingResolutionSpectralCube method), 197
SpectralCube (class in spectral_cube), 162
squeeze() (spectral_cube.OneDSpectrum method), 80
squeeze() (spectral_cube.Projection method), 117
squeeze() (spectral_cube.Slice method), 154
std() (spectral_cube.OneDSpectrum method), 80
std() (spectral_cube.Projection method), 117
std() (spectral_cube.Slice method), 154
std() (spectral_cube.VaryingResolutionSpectralCube method), 197

StokesSpectralCube (class in spectral_cube), 180
strides (numpy.core._internal.OneDSpectrum._ctypes attribute), 54
strides (numpy.core._internal.Projection._ctypes attribute), 91
strides (numpy.core._internal.Slice._ctypes attribute), 129
strides (spectral_cube.OneDSpectrum attribute), 59
strides (spectral_cube.Projection attribute), 97
strides (spectral_cube.Slice attribute), 134
strides_as() (numpy.core._internal.OneDSpectrum._ctypes method), 55
strides_as() (numpy.core._internal.Projection._ctypes method), 92
strides_as() (numpy.core._internal.Slice._ctypes method), 129
subcube() (spectral_cube.SpectralCube method), 176
subcube() (spectral_cube.SpectralCube method), 176
subcube_from_crtfregion() (spectral_cube.SpectralCube method), 176
subcube_from_crtfregion() (spectral_cube.VaryingResolutionSpectralCube method), 198
subcube_from_ds9region() (spectral_cube.SpectralCube method), 177
subcube_from_ds9region() (spectral_cube.SpectralCube method), 198
subcube_from_mask() (spectral_cube.SpectralCube method), 177
subcube_from_mask() (spectral_cube.VaryingResolutionSpectralCube method), 198
subcube_from_regions() (spectral_cube.SpectralCube method), 177
subcube_from_regions() (spectral_cube.VaryingResolutionSpectralCube method), 198
subcube_slices_from_mask() (spectral_cube.SpectralCube method), 177
subcube_slices_from_mask() (spectral_cube.VaryingResolutionSpectralCube method), 199
subimage() (spectral_cube.lower_dimensional_structures.LowerDimensionalObject method), 207
subimage() (spectral_cube.Projection method), 117
subimage() (spectral_cube.Slice method), 155
sum() (spectral_cube.OneDSpectrum method), 80
sum() (spectral_cube.Projection method), 117
sum() (spectral_cube.Slice method), 155
sum() (spectral_cube.SpectralCube method), 177
sum() (spectral_cube.VaryingResolutionSpectralCube method), 199
swapaxes() (spectral_cube.OneDSpectrum method), 80
swapaxes() (spectral_cube.Projection method), 118
swapaxes() (spectral_cube.Slice method), 155

t (spectral_cube.OneDSpectrum attribute), 53
t (spectral_cube.Projection attribute), 90
t (spectral_cube.Slice attribute), 127
take() (spectral_cube.OneDSpectrum method), 80
take() (spectral_cube.Projection method), 118

take() (spectral_cube.Slice method), 155
test() (in module spectral_cube), 37
to() (spectral_cube.lower_dimensional_structures.LowerDimensionalObject method), 206
to() (spectral_cube.lower_dimensional_structures.Projection method), 208
to() (spectral_cube.OneDSpectrum method), 80
to() (spectral_cube.Projection method), 118
to() (spectral_cube.Slice method), 155

to() (spectral_cube.SpectralCube method), 178
to() (spectral_cube.VaryingResolutionSpectralCube method), 199
to_ds9() (spectral_cube.SpectralCube method), 178
to_ds9() (spectral_cube.VaryingResolutionSpectralCube method), 199
to_glue() (spectral_cube.SpectralCube method), 178
to_glue() (spectral_cube.VaryingResolutionSpectralCube method), 199
to_pvextractor() (spectral_cube.SpectralCube method), 178
to_pvextractor() (spectral_cube.VaryingResolutionSpectralCube method), 200
to_string() (spectral_cube.OneDSpectrum method), 81
to_string() (spectral_cube.Projection method), 118
to_string() (spectral_cube.Slice method), 155
to_value() (spectral_cube.OneDSpectrum method), 81
to_value() (spectral_cube.Projection method), 119
to_value() (spectral_cube.Slice method), 156
to_yt() (spectral_cube.SpectralCube method), 178
to_yt() (spectral_cube.VaryingResolutionSpectralCube method), 200
tobytes() (spectral_cube.OneDSpectrum method), 82
tobytes() (spectral_cube.Projection method), 119

tobytes() (spectral_cube.Slice method), 156
tofile() (spectral_cube.OneDSpectrum method), 82
tofile() (spectral_cube.Projection method), 120
tolist() (spectral_cube.Slice method), 157
tolist() (spectral_cube_OneDSpectrum method), 83
tolist() (spectral_cube_Projection method), 120
tolist() (spectral_cube.Slice method), 158
tostring() (spectral_cube.OneDSpectrum method), 83
tostring() (spectral_cube.Projection method), 121
tostring() (spectral_cube.Slice method), 158
trace() (spectral_cube.OneDSpectrum method), 84
trace() (spectral_cube.Projection method), 121	race() (spectral_cube.Slice method), 159
transpose() (spectral_cube.OneDSpectrum method), 84
transpose() (spectral_cube.Projection method), 121
transpose() (spectral_cube.Slice method), 159

U
unmasked_copy() (spectral_cube.SpectralCube method), 179
unmasked_copy() (spectral_cube.VaryingResolutionSpectralCube method), 200

V
var() (spectral_cube.OneDSpectrum method), 85
var() (spectral_cube.Projection method), 122
var() (spectral_cube.Slice method), 160
VaryingResolutionSpectralCube (class in spectral_cube), 182
view() (spectral_cube.BooleanArrayMask method), 40
view() (spectral_cube.CompositeMask method), 42
view() (spectral_cube.FunctionMask method), 43
view() (spectral_cube.InvertedMask method), 45
view() (spectral_cube.LazyComparisonMask method), 47
view() (spectral_cube.LazyMask method), 48
view() (spectral_cube.MaskBase method), 50
view() (spectral_cube.OneDSpectrum method), 85
view() (spectral_cube.Projection method), 122
view() (spectral_cube.Slice method), 160

W
with_beam() (spectral_cube.lower_dimensional_structures.Slice method), 208
with_beam() (spectral_cube.lower_dimensional_structures.Projection method), 208
with_beam() (spectral_cube.OneDSpectrum method), 87
with_beam() (spectral_cube.Projection method), 124
with_beam() (spectral_cube.Slice method), 161
with_beam() (spectral_cube.SpectralCube method), 179
with_beams() (spectral_cube.VaryingResolutionSpectralCube method), 200

with_fill_value() (spectral_cube.lower_dimensional_structures.Projection method), 208
with_fill_value() (spectral_cube.OneDSpectrum method), 87
with_fill_value() (spectral_cube.Projection method), 124
with_fill_value() (spectral_cube.Slice method), 162
with_fill_value() (spectral_cube.SpectralCube method), 179
with_fill_value() (spectral_cube.VaryingResolutionSpectralCube method), 200
with_mask() (spectral_cube.SpectralCube method), 179
with_mask() (spectral_cube.StokesSpectralCube method), 181
with_mask() (spectral_cube.VaryingResolutionSpectralCube method), 201
with_spectral_unit() (spectral_cube.BooleanArrayMask method), 40
with_spectral_unit() (spectral_cube.CompositeMask method), 42
with_spectral_unit() (spectral_cube.FunctionMask method), 44
with_spectral_unit() (spectral_cube.InvertedMask method), 45
with_spectral_unit() (spectral_cube.LazyComparisonMask method), 47
with_spectral_unit() (spectral_cube.LazyMask method), 48
with_spectral_unit() (spectral_cube.OneDSpectrum method), 87
with_spectral_unit() (spectral_cube.SpectralCube method), 180
with_spectral_unit() (spectral_cube.StokesSpectralCube method), 181
with_spectral_unit() (spectral_cube.VaryingResolutionSpectralCube method), 181
world2yt() (spectral_cube.ytcube.ytCube method), 204
world_spines() (spectral_cube.Projection method), 124
world_spines() (spectral_cube.Slice method), 162
world_spines() (spectral_cube.SpectralCube method), 180
write() (spectral_cube.lower_dimensional_structures.LowerDimensionalObject method), 206
write() (spectral_cube.OneDSpectrum method), 87
write() (spectral_cube.Projection method), 124
write() (spectral_cube.Slice method), 162
write() (spectral_cube.SpectralCube method), 180
write() (spectral_cube.StokesSpectralCube method), 182
write() (spectral_cube.VaryingResolutionSpectralCube method), 202

Y

yt2world() (spectral_cube.ytcube.ytCube method), 204
ytCube (class in spectral_cube.ytcube), 202