# CONTENTS

1 Array objects 3
  1.1 The N-dimensional array (*ndarray*) 3
  1.2 Scalars 39
  1.3 Data type objects (*dtype*) 45
  1.4 Indexing 57
  1.5 Standard array subclasses 61
  1.6 Masked arrays 76
  1.7 The Array Interface 263

2 Universal functions (*ufunc*) 269
  2.1 Broadcasting 269
  2.2 Output type determination 270
  2.3 Use of internal buffers 270
  2.4 Error handling 270
  2.5 Casting Rules 273
  2.6 *ufunc* 275
  2.7 Available *ufuncs* 283

3 Routines 287
  3.1 Array creation routines 287
  3.2 Array manipulation routines 318
  3.3 Indexing routines 353
  3.4 Data type routines 377
  3.5 Input and output 389
  3.6 Discrete Fourier Transform (*numpy.fft*) 409
  3.7 Linear algebra (*numpy.linalg*) 429
  3.8 Random sampling (*numpy.random*) 457
  3.9 Sorting and searching 507
  3.10 Logic functions 520
  3.11 Binary operations 537
  3.12 Statistics 544
  3.13 Mathematical functions 565
  3.14 Functional programming 625
  3.15 Polynomials 630
  3.16 Financial functions 643
  3.17 Set routines 651
  3.18 Window functions 656
  3.19 Floating point error handling 666
  3.20 Masked array operations 673
Release
1.4

Date
April 22, 2010

This reference manual details functions, modules, and objects included in Numpy, describing what they are and what they do. For learning how to use NumPy, see also NumPy User Guide (in NumPy User Guide).
NumPy provides an N-dimensional array type, the *ndarray*, which describes a collection of “items” of the same type. The items can be indexed using for example N integers.

All ndarrays are *homogenous*: every item takes up the same size block of memory, and all blocks are interpreted in exactly the same way. How each item in the array is to be interpreted is specified by a separate *data-type object*, one of which is associated with every array. In addition to basic types (integers, floats, *etc.*), the data type objects can also represent data structures.

An item extracted from an array, *e.g.*, by indexing, is represented by a Python object whose type is one of the *array scalar types* built in Numpy. The array scalars allow easy manipulation of also more complicated arrangements of data.

**Figure 1.1:** Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type object that describes the layout of a single fixed-size element of the array, 3) the array-scalar Python object that is returned when a single element of the array is accessed.

### 1.1 The N-dimensional array (*ndarray*)

An *ndarray* is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its *shape*, which is a *tuple* of *N* positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate *data-type object (dtype)*, one of which is associated with each ndarray.

As with other container objects in Python, the contents of an *ndarray* can be accessed and modified by *indexing or slicing* the array (using, for example, *N* integers), and via the methods and attributes of the *ndarray*. Different
ndarrays can share the same data, so that changes made in one ndarray may be visible in another. That is, an ndarray can be a “view” to another ndarray, and the data it is referring to is taken care of by the “base” ndarray. ndarrays can also be views to memory owned by Python strings or objects implementing the buffer or array interfaces.

**Example**

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```python
g = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
g.dtype
```

The array can be indexed using Python container-like syntax:

```python
g[1,2]  # i.e., the element of g in the second row, third column
```

For example slicing can produce views of the array:

```python
g[:,1]
g[0] = 9  # this also changes the corresponding element in g
g
```

### 1.1.1 Constructing arrays

New arrays can be constructed using the routines detailed in *Array creation routines*, and also by using the low-level ndarray constructor:

```python
d = np.array([1, 2, 3, 4], dtype='int32')
d
```

### Notes

- **shape**
  - A tuple of ints indicating the size of the array in each dimension.

- **dtype**
  - A string representing the data type and its formatting.

- **class ndarray**
  - An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

- **Parameters**
  - For the __new__ method (see Notes above):
    - shape: tuple of ints
Shape of created array.

dtype : data-type, optional
    Any object that can be interpreted as a numpy data type.

buffer : object exposing buffer interface, optional
    Used to fill the array with data.

offset : int, optional
    Offset of array data in buffer.

strides : tuple of ints, optional
    Strides of data in memory.

order : {'C', 'F'}, optional
    Row-major or column-major order.

See Also:

array
    Construct an array.

zeros
    Create an array, each element of which is zero.

empty
    Create an array, but leave its allocated memory unchanged (i.e., it contains "garbage").

dtype
    Create a data-type.

Notes

There are two modes of creating an array using __new__:

1. If buffer is None, then only shape, dtype, and order are used.
2. If buffer is an object exposing the buffer interface, then all keywords are interpreted.

No __init__ method is needed because the array is fully initialized after the __new__ method.

Examples

These examples illustrate the low-level ndarray constructor. Refer to the See Also section above for easier ways of constructing an ndarray.

First mode, buffer is None:

    >>> np.ndarray(shape=(2,2), dtype=float, order='F')
    array([[ -1.13698227e+002, 4.25087011e-303],
    [ 2.88528414e-306, 3.27025015e-309]])

Second mode:

    >>> np.ndarray((2,), buffer=np.array([1,2,3]),
    ...             offset=np.int_().itemsize,
    ...             dtype=int) # offset = 1*itemsize, i.e. skip first element
    array([2, 3])
1.1.2 Indexing arrays

Arrays can be indexed using an extended Python slicing syntax, `array[selection]`. Similar syntax is also used for accessing fields in a record array.

See Also:
Array Indexing.

1.1.3 Internal memory layout of an ndarray

An instance of class `ndarray` consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps `N` integers into the location of an item in the block. The ranges in which the indices can vary is specified by the `shape` of the array. How many bytes each item takes and how the bytes are interpreted is defined by the `data-type object` associated with the array.

A segment of memory is inherently 1-dimensional, and there are many different schemes for arranging the items of an `N`-dimensional array in a 1-dimensional block. Numpy is flexible, and `ndarray` objects can accommodate any strided indexing scheme. In a strided scheme, the `N`-dimensional index `(n_0, n_1, ..., n_{N-1})` corresponds to the offset (in bytes)

\[ n_{\text{offset}} = \sum_{k=0}^{N-1} s_k n_k \]

from the beginning of the memory block associated with the array. Here, `s_k` are integers which specify the strides of the array. The column-major order (used, for example, in the Fortran language and in Matlab) and row-major order (used in C) schemes are just specific kinds of strided scheme, and correspond to the strides:

\[ s_{\text{column}}^k = \prod_{j=0}^{k-1} d_j, \quad s_{\text{row}}^k = \prod_{j=k+1}^{N-1} d_j. \]

where \( d_j = \text{self.itemsize} \times \text{self.shape}[j] \).

Both the C and Fortran orders are contiguous, i.e., single-segment, memory layouts, in which every part of the memory block can be accessed by some combination of the indices.

Data in new `ndarrays` is in the row-major (C) order, unless otherwise specified, but, for example, basic array slicing often produces views in a different scheme.

Note: Several algorithms in NumPy work on arbitrarily strided arrays. However, some algorithms require single-segment arrays. When an irregularly strided array is passed in to such algorithms, a copy is automatically made.

1.1.4 Array attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed
attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array. Information on each attribute is given below.

Memory layout

The following attributes contain information about the memory layout of the array:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.flags</td>
<td>Information about the memory layout of the array.</td>
</tr>
<tr>
<td>ndarray.shape</td>
<td>Tuple of array dimensions.</td>
</tr>
<tr>
<td>ndarray.strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</td>
</tr>
<tr>
<td>ndarray.ndim</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>ndarray.data</td>
<td>Python buffer object pointing to the start of the array’s data.</td>
</tr>
<tr>
<td>ndarray.size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>ndarray.itemsize</td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td>ndarray.nbytes</td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td>ndarray.base</td>
<td>Base object if memory is from some other object.</td>
</tr>
</tbody>
</table>

Notes

The flags object can be accessed dictionary-like (as in a.flags[‘WRITEABLE’]), or by using lowercased attribute names (as in a.flags.writeable). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling ndarray.setflags.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set False.
- ALIGNED can only be set True if the data is truly aligned.
- WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Attributes

shape

Tuple of array dimensions.

Notes

May be used to “reshape” the array, as long as this would not require a change in the total number of elements

Examples

```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.]])
```
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:

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

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):

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

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
generate code examples here
```
**ndim**
Number of array dimensions.

**Examples**

```python
generate-solutions()
```

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

**size**
Number of elements in the array.
Equivalent to `np.prod(a.shape)`, i.e., the product of the array’s dimensions.

**Examples**

```python
generate-solutions()
```

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

**Examples**

```python
generate-solutions()
```

**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
generate-solutions()
```

**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
```

Note: XXX: update and check these docstrings.

Data type

See Also:

Data type objects

The data type object associated with the array can be found in the `dtype` attribute:

```python
ndarray.dtype  # Data-type of the array’s elements.
```

```
astype
```

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]])
```

```python
>>> x.dtype
dtype('int32')
```

```python
>>> type(x.dtype)
<type 'numpy.dtype'>
```

Note: XXX: update the dtype attribute docstring: setting etc.

Other attributes

```python
ndarray.T
ndarray.real
ndarray.imag
ndarray.flat
ndarray.ctypes
__array_priority__
```

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

The real part of the array.

The imaginary part of the array.

A 1-D iterator over the array.

An object to simplify the interaction of the array with the ctypes module.
\texttt{T}

Same as \texttt{self.transpose()}, except that \texttt{self} is returned if \texttt{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.])
```

\texttt{real}

The real part of the array.

**See Also:**

\texttt{numpy.real}

equivalent function

**Examples**

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

\texttt{imag}

The imaginary part of the array.

**Examples**

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

\texttt{flat}

A 1-D iterator over the array.

This is a \texttt{numpy.flatiter} instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

**See Also:**

\texttt{flatten}

Return a copy of the array collapsed into one dimension.
Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

ctypes

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

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters

None:

Returns

c : Python object

Possessing attributes data, shape, strides, etc.

See Also:

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype(‘p’) on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
•strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

•data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

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

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

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```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>
```

Array interface

See Also:

The Array Interface.
numarray.ctypes
An object to simplify the interaction of the array with the ctypes module.

cylotypes
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 cylotypes objects that can be used as arguments to a shared library.

Parameters
None:

Returns
c : Python object
Possessing attributes data, shape, strides, etc.

See Also:
numpy.ctypeslib

Notes
Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

• data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

• shape (c_intp*self.ndim): A cylotypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The cylotypes array contains the shape of the underlying array.

• strides (c_intp*self.ndim): A cylotypes array of length self.ndim where the basetype is the same as for the shape attribute. This cylotypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

• data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(cylotypes.c_void_p). Perhaps you want to use the data as a pointer to a cylotypes array of floating-point data: self.data.as(cylotypes.POINTER(cylotypes.c_double)).

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

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

Be careful using the cylotypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(cylotypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.
If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x
darray([[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>
```

Note: XXX: update and check these docstrings.

1.1.5 Array methods

An ndarray object has many methods which operate on or with the array in some fashion, typically returning an array result. These methods are briefly explained below. (Each method’s docstring has a more complete description.)

For the following methods there are also corresponding functions in numpy: all, any, argmax, argmin, argsort, choose, clip, compress, copy, cumprod, cumsum, diagonal, imag, max, mean, min, nonzero, prod, ptp, put, ravel, real, repeat, reshape, round, searchsorted, sort, squeeze, std, sum, swapaxes, take, trace, transpose, var.
## Array conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.item(*args)</td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td>ndarray.tolist()</td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td>ndarray.itemset()</td>
<td></td>
</tr>
<tr>
<td>ndarray.tostring()</td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td>ndarray.tolist()</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>ndarray.dumps()</td>
<td></td>
</tr>
<tr>
<td>ndarray.dump()</td>
<td>Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpypick.load.</td>
</tr>
<tr>
<td>ndarray.astype()</td>
<td>Returns the pickle of the array as a string. pickle.loads or numpypick.loads will convert the string back to an array.</td>
</tr>
<tr>
<td>ndarray.fill()</td>
<td></td>
</tr>
<tr>
<td>ndarray.getfield()</td>
<td>Copy of the array, cast to a specified type.</td>
</tr>
<tr>
<td>ndarray.setflags()</td>
<td>Swap the bytes of the array elements</td>
</tr>
<tr>
<td>ndarray.copy()</td>
<td>Return a copy of the array.</td>
</tr>
<tr>
<td>ndarray.view()</td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td>ndarray.getfield()</td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td>ndarray.setflags()</td>
<td></td>
</tr>
</tbody>
</table>

### item()

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 \(a[\text{args}]\), 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],
       [7, 0, 3],
       [6, 4, 2]])
```
```python
>>> x = np.array([[2, 8, 3], [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

**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]
```

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

**itemset()**

**tostring()**

Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string 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.

**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 : str

A Python string exhibiting a copy of a’s raw data.
Examples

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

tofile()

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.tostring())`.
- `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.

dump()

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`

astype()

Copy of the array, cast to a specified type.

Parameters
- `t` : string or dtyple
  Typecode or data-type to which the array is cast.
Examples

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

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

`byteswap()`

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(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')
```

`copy()`

Return a copy of the array.

**Parameters**

- `order` : {'C', 'F', 'A'}, optional

  By default, the result is stored in C-contiguous (row-major) order in memory. If order is F, the result has 'Fortran' (column-major) order. If order is 'A' ('Any'), then the result has the same order as the input.

Examples

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

>>> y = x.copy()
```
>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

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

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

**view()**

New view of array with the same data.

**Parameters**

- **dtype**: data-type
  
  Data-type descriptor of the returned view, e.g. float32 or int16.

- **type**: python type
  
  Type of the returned view, e.g. ndarray or matrix.

**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 dtype. 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.

**Examples**

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

Viewing array data using a different type and dtype:

>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print type(y)
<class 'numpy.matrixlib.defmatrix.matrix'>

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

```python
```
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 record array:

```python
>>> z = x.view(np.recarray)
>>> z[0]
array([1,], dtype=int8)
```

Views share data:

```python
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

**getfield()**

Returns a field of the given array as a certain type.

A field is a view of the array data with each itemsize determined by the given type and the offset into the current array, i.e. from `offset * dtype.itemsize` to `(offset+1) * dtype.itemsize`.

**Parameters**

`dtype`: str

String denoting the data type of the field.

`offset`: int

Number of `dtype.itemsize`’s to skip before beginning the element view.

**Examples**

```python
>>> x = np.diag([1.+1.j]*2)
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 1.+1.j]])
>>> x.dtype
dtype('complex128')

>>> x.getfield('complex64', 0) # Note how this != x
array([[ 0.+1.875j, 0.+0.j],
       [ 0.+0.j, 0.+1.875j]], dtype=complex64)

>>> x.getfield('complex64',1) # Note how different this is than x
array([[ 0. +5.87173204e-39j, 0. +0.00000000e+00j],
       [ 0. +0.00000000e+00j, 0. +5.87173204e-39j]], dtype=complex64)

>>> x.getfield('complex128', 0) # == x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 1.+1.j]])
```

If the argument `dtype` is the same as `x.dtype`, then `offset != 0` raises a `ValueError`:
setflags()

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by a (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional
Describes whether or not a can be written to.

align : bool, optional
Describes whether or not a is aligned properly for its type.

uic : bool, optional
Describes whether or not a is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

Examples

```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
UPDATEIFCOPY : False

```python
>>> y.setflags(write=0, align=0)
>>> y.flags
    C_CONTIGUOUS : True
    F_CONTIGUOUS : False
    OWNDATA : True
    WRITEABLE : False
    ALIGNED : False
    UPDATEIFCOPY : False
>>> y.setflags(uic=1)
```

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

`fill()`

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.])
```

**Note**: XXX: update and check these docstrings.

### Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with `n` integers which will be interpreted as an `n`-tuple.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ndarray.reshape(shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape. Change shape and size of array in-place. Returns a view of the array with axes transposed. Return a view of the array with <code>axis1</code> and <code>axis2</code> interchanged. Return a copy of the array collapsed into one dimension. Return a flattened array. Remove single-dimensional entries from the shape of <code>a</code>.</td>
</tr>
<tr>
<td><code>ndarray.resize(new_shape[, refcheck, order])</code></td>
<td></td>
</tr>
<tr>
<td><code>ndarray.transpose(*axes)</code></td>
<td></td>
</tr>
<tr>
<td><code>ndarray.swapaxes(axis1, axis2)</code></td>
<td></td>
</tr>
<tr>
<td><code>ndarray.flatten([order])</code></td>
<td></td>
</tr>
<tr>
<td><code>ndarray.ravel([order])</code></td>
<td></td>
</tr>
<tr>
<td><code>ndarray.squeeze()</code></td>
<td></td>
</tr>
</tbody>
</table>

**reshape**

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

**See Also:**

1.1. The N-dimensional array (`ndarray`)
`numpy.reshape` equivalent function

`resize()`  
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.
- `order` : bool, do not use.  
  A SystemError is raised when this parameter is specified.

Returns
- `None` : 

Raises
- `ValueError` :  
  If `a` does not own its own data or references or views to it exist, and the data memory must be changed.
- `SystemError` : 
  If the `order` keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

`resize`  
Return a new array with the specified shape.

Notes
This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set `refcheck` to False.

Examples
Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]])
>>> print a.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> a.resize((2, 1))
>>> a
```
array( [[0],
       [1]])

>>> a = np.array( [[0, 1], [2, 3]], order='F')
>>> print a.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> a.resize((2, 1))
>>> a
array( [[0],
       [2]])

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

>>> b = np.array( [[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
>>> b
array( [[0, 1, 2],
       [3, 0, 0]])

Referencing an array prevents resizing...

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

Unless refcheck is False:

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

transpose()
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]])
```

**swapaxes()**

Return a view of the array with \(axis1\) and \(axis2\) interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See Also:**

*`numpy.swapaxes`*  
equivalent function

**flatten()**

Return a copy of the array collapsed into one dimension.

**Parameters**

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

Whether to flatten in C (row-major) or Fortran (column-major) order. 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])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

`ravel()`

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.

`squeeze()`

Remove single-dimensional entries from the shape of `a`.

Refer to `numpy.squeeze` for full documentation.

See Also:

`numpy.squeeze`  
equivalent function

Item selection and manipulation

For array methods that take an `axis` keyword, it defaults to `None`. If axis is `None`, then the array is treated as a 1-D array. Any other value for `axis` represents the dimension along which the operation should proceed.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ndarray.take(indices[, axis, out, mode])</code></td>
<td>Return an array formed from the elements of <code>a</code> at the given indices.</td>
</tr>
<tr>
<td><code>ndarray.put(indices, values[, out])</code></td>
<td>Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.</td>
</tr>
<tr>
<td><code>ndarray.repeat(repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>ndarray.choose(choices[, out, mode])</code></td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td><code>ndarray.sort([axis, kind, order])</code></td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td><code>ndarray.argsort([axis, kind, order])</code></td>
<td>Returns the indices that would sort this array.</td>
</tr>
<tr>
<td><code>ndarray.searchsorted(v[, side])</code></td>
<td>Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.</td>
</tr>
<tr>
<td><code>ndarray.insert(condition[, axis, out])</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>ndarray.compress(condition[, axis])</code></td>
<td>Return selected slices of this array along given axis.</td>
</tr>
<tr>
<td><code>ndarray.diagonal([offset, axis1, axis2])</code></td>
<td>Return specified diagonals.</td>
</tr>
</tbody>
</table>

`take()`

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

**put()**

Set a.flat[n] = values[n] for all n in indices.

Refer to `numpy.put` for full documentation.

**See Also:**

`numpy.put` equivalent function

**repeat()**

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See Also:**

`numpy.repeat` equivalent function

**choose()**

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

**sort()**

Sort an array, in-place.

**Parameters**

- **axis**: int, optional
  - Axis along which to sort. Default is -1, which means sort along the last axis.

- **kind**: {'quicksort', 'mergesort', 'heapsort'}, optional
  - Sorting algorithm. Default is 'quicksort'.

- **order**: list, optional
  - When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**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.

**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],
       [3, 1]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [4, 1]])
```

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', '<i4')])
```

**argsort()**

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

**See Also:**

`numpy.argsort`

equivalent function

**searchsorted()**

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

**nonzero()**

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

Refer to `numpy.nonzero` for full documentation.

**See Also:**

`numpy.nonzero`

equivalent function

1.1. The N-dimensional array (`ndarray`)
compress()
Return selected slices of this array along given axis.
Refer to numpy.compress for full documentation.
See Also:

compress
equivalent function

diagonal()
Return specified diagonals.
Refer to numpy.diagonal for full documentation.
See Also:

diagonal
equivalent function

Calculation

Many of these methods take an argument named axis. In such cases,

- If axis is None (the default), the array is treated as a 1-D array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. (An array scalar is an instance of the types/classes float32, float64, etc., whereas a 0-dimensional array is an ndarray instance containing precisely one array scalar.)

- If axis is an integer, then the operation is done over the given axis (for each 1-D subarray that can be created along the given axis).

Example of the axis argument

A 3-dimensional array of size 3 x 3 x 3, summed over each of its three axes

```python
clamp = [[[ 0, 1, 2],
    [ 3, 4, 5],
    [ 6, 7, 8]],
[[ 9, 10, 11],
[12, 13, 14],
[15, 16, 17]],
[[18, 19, 20],
[21, 22, 23],
[24, 25, 26]]
```
The parameter `dtype` specifies the data type over which a reduction operation (like summing) should take place. The default reduce data type is the same as the data type of `self`. To avoid overflow, it can be useful to perform the reduction using a larger data type.

For several methods, an optional `out` argument can also be provided and the result will be placed into the output array given. The `out` argument must be an `ndarray` and have the same number of elements. It can have a different data type in which case casting will be performed.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argmax()</code></td>
<td>Return indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td></td>
<td>Refer to <code>numpy.argmax</code> for full documentation.</td>
</tr>
<tr>
<td><strong>See Also:</strong></td>
<td></td>
</tr>
<tr>
<td><code>numpy.argmax</code></td>
<td>equivalent function</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td></td>
<td>Refer to <code>numpy.amin</code> for full documentation.</td>
</tr>
<tr>
<td><strong>See Also:</strong></td>
<td></td>
</tr>
<tr>
<td><code>numpy.amin</code></td>
<td>equivalent function</td>
</tr>
<tr>
<td><code>argmin()</code></td>
<td>Return indices of the minimum values along the given axis of <code>a</code>.</td>
</tr>
<tr>
<td></td>
<td>Refer to <code>numpy.argmin</code> for detailed documentation.</td>
</tr>
<tr>
<td><strong>See Also:</strong></td>
<td></td>
</tr>
</tbody>
</table>

1.1. The N-dimensional array (`ndarray`) 31
numpy.argmin
   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

clip()
   Return an array whose values are limited to [a_min, a_max].
   Refer to numpy.clip for full documentation.
   See Also:

   numpy.clip
   equivalent function

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

   numpy.conjugate
   equivalent function

round()
   Return an array rounded a to the given number of decimals.
   Refer to numpy.around for full documentation.
   See Also:

   numpy.around
   equivalent function

trace()
   Return the sum along diagonals of the array.
   Refer to numpy.trace for full documentation.
   See Also:

   numpy.trace
   equivalent function

sum()
   Return the sum of the array elements over the given axis.
   Refer to numpy.sum for full documentation.
   See Also:
numpy.sum
   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

mean()
   Returns the average of the array elements along given axis.
   Refer to numpy.mean for full documentation.

   See Also:

   numpy.mean
      equivalent function

var()
   Returns the variance of the array elements, along given axis.
   Refer to numpy.var for full documentation.

   See Also:

   numpy.var
      equivalent function

std()
   Returns the standard deviation of the array elements along given axis.
   Refer to numpy.std for full documentation.

   See Also:

   numpy.std
      equivalent function

prod()
   Return the product of the array elements over the given axis
   Refer to numpy.prod for full documentation.

   See Also:

   numpy.prod
      equivalent function

cumprod()
   Return the cumulative product of the elements along the given axis.
   Refer to numpy.cumprod for full documentation.

   See Also:
numpy.cumprod
equivalent function

all()
Returns True if all elements evaluate to True.
Refer to numpy.all for full documentation.

See Also:

numpy.all
equivalent function

any()
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

1.1.6 Arithmetic and comparison operations

Note: XXX: write all attributes explicitly here instead of relying on the auto* stuff? Arithmetic and comparison operations on ndarrays are defined as element-wise operations, and generally yield ndarray objects as results.

Each of the arithmetic operations (+, -, *, /, //, %, divmod(), **, or pow(), <, >, <=, >=, ~, |, ^, &) and the comparisons (==, <, >, <=, >=, !~) is equivalent to the corresponding universal function (or ufunc for short) in Numpy. For more information, see the section on Universal Functions.

Comparison operators:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.<strong>lt</strong>()</td>
<td>x.<strong>lt</strong>(y) &lt;==&gt; x&lt;y</td>
</tr>
<tr>
<td>ndarray.<strong>le</strong>()</td>
<td>x.<strong>le</strong>(y) &lt;==&gt; x&lt;=y</td>
</tr>
<tr>
<td>ndarray.<strong>gt</strong>()</td>
<td>x.<strong>gt</strong>(y) &lt;==&gt; x&gt;y</td>
</tr>
<tr>
<td>ndarray.<strong>ge</strong>()</td>
<td>x.<strong>ge</strong>(y) &lt;==&gt; x&gt;=y</td>
</tr>
<tr>
<td>ndarray.<strong>eq</strong>()</td>
<td>x.<strong>eq</strong>(y) &lt;==&gt; x==y</td>
</tr>
<tr>
<td>ndarray.<strong>ne</strong>()</td>
<td>x.<strong>ne</strong>(y) &lt;==&gt; x!=y</td>
</tr>
</tbody>
</table>

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

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

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

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

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

__ne__()
 x.__ne__(y) <==> x!=y
Truth value of an array (bool):

```python
ndarray.__nonzero__()  x.__nonzero__() == x != 0
__nonzero__()
    x.__nonzero__() == x != 0
```

**Note:** Truth-value testing of an array invokes `ndarray.__nonzero__`, which raises an error if the number of elements in the array is larger than 1, because the truth value of such arrays is ambiguous. Use `.any()` and `.all()` instead to be clear about what is meant in such cases. (If the number of elements is 0, the array evaluates to `False`.)

Unary operations:

```python
ndarray.__neg__()  x.__neg__() == -x
ndarray.__pos__()  x.__pos__() == +x
ndarray.__abs__()  x.__abs__() == abs(x)
ndarray.__invert__()  x.__invert__() == ~x
```

Arithmetic:

```python
ndarray.__add__()  x.__add__(y) == x+y
ndarray.__sub__()  x.__sub__(y) == x-y
ndarray.__mul__()  x.__mul__(y) == x*y
ndarray.__div__()  x.__div__(y) == x/y
ndarray.__truediv__()  x.__truediv__(y) == x/y
ndarray.__floordiv__()  x.__floordiv__(y) == x//y
ndarray.__mod__()  x.__mod__(y) == x%y
ndarray.__divmod__(y)  x.__divmod__(y) == (x//y, x%y)
ndarray.__pow__(y[, z])  x.__pow__(y[, z]) == x**y
ndarray.__lshift__()  x.__lshift__(y) == x<<y
ndarray.__rshift__()  x.__rshift__(y) == x>>y
ndarray.__and__()  x.__and__(y) == x&y
ndarray.__or__()  x.__or__(y) == x|y
ndarray.__xor__()  x.__xor__(y) == x^y
```

1.1. The N-dimensional array (`ndarray`)
__floordiv__()  
x.__floordiv__(y) <==> x//y

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

__divmod__()  

__pow__()  

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

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

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

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

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

Note:

• Any third argument to pow is silently ignored, as the underlying ufunc takes only two arguments.

• The three division operators are all defined; div is active by default, truediv is active when __future__ division is in effect.

• Because ndarray is a built-in type (written in C), the __r{op}__ special methods are not directly defined.

• The functions called to implement many arithmetic special methods for arrays can be modified using set_numeric_ops.

Arithmetic, in-place:

<table>
<thead>
<tr>
<th>ndarray.<strong>iadd</strong>()</th>
<th>x.<strong>iadd</strong>(y) &lt;==&gt; x+y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.<strong>isub</strong>()</td>
<td>x.<strong>isub</strong>(y) &lt;==&gt; x-y</td>
</tr>
<tr>
<td>ndarray.<strong>imul</strong>()</td>
<td>x.<strong>imul</strong>(y) &lt;==&gt; x*y</td>
</tr>
<tr>
<td>ndarray.<strong>idiv</strong>()</td>
<td>x.<strong>idiv</strong>(y) &lt;==&gt; x/y</td>
</tr>
<tr>
<td>ndarray.<strong>itruediv</strong>()</td>
<td>x.<strong>itruediv</strong>(y) &lt;==&gt; x/y</td>
</tr>
<tr>
<td>ndarray.<strong>ifloordiv</strong>()</td>
<td>x.<strong>ifloordiv</strong>(y) &lt;==&gt; x//y</td>
</tr>
<tr>
<td>ndarray.<strong>imod</strong>()</td>
<td>x.<strong>imod</strong>(y) &lt;==&gt; x%y</td>
</tr>
<tr>
<td>ndarray.<strong>ipow</strong>()</td>
<td>x.<strong>ipow</strong>(y) &lt;==&gt; x**y</td>
</tr>
<tr>
<td>ndarray.<strong>ilshift</strong>()</td>
<td>x.<strong>ilshift</strong>(y) &lt;==&gt; x&lt;&lt;y</td>
</tr>
<tr>
<td>ndarray.<strong>irshift</strong>()</td>
<td>x.<strong>irshift</strong>(y) &lt;==&gt; x&gt;&gt;y</td>
</tr>
<tr>
<td>ndarray.<strong>iand</strong>()</td>
<td>x.<strong>iand</strong>(y) &lt;==&gt; x&amp;y</td>
</tr>
<tr>
<td>ndarray.<strong>ior</strong>()</td>
<td>x.<strong>ior</strong>(y) &lt;==&gt; x</td>
</tr>
<tr>
<td>ndarray.<strong>ixor</strong>()</td>
<td>x.<strong>ixor</strong>(y) &lt;==&gt; x^y</td>
</tr>
</tbody>
</table>

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

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

__imul__()  
x.__imul__(y) <==> x*y
Warning: In place operations will perform the calculation using the precision decided by the data type of the two operands, but will silently downcast the result (if necessary) so it can fit back into the array. Therefore, for mixed precision calculations, A {op}= B can be different than A = A {op} B. For example, suppose a = ones((3,3)). Then, a += 3j is different than a = a + 3j: while they both perform the same computation, a += 3 casts the result to fit back in a, whereas a = a + 3j re-binds the name a to the result.

1.1.7 Special methods

For standard library functions:

```plaintext
ndarray.__copy__(order)  # Return a copy of the array.
ndarray.__deepcopy__()   # a.__deepcopy__() -> Deep copy of array.
ndarray.__reduce__()     # For pickling.
ndarray.__setstate__(version, shape, dtype, ...) # For unpickling.
```

__copy__()  
Return a copy of the array.

Parameters

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

  If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

__deepcopy__()  
A.__deepcopy__() -> Deep copy of array.

Used if copy.deepcopy is called on an array.
NumPy Reference, Release 1.4.1

__reduce__()  
For pickling.

__setstate__()  
For unpickling.

**Parameters**

- **version**: int  
  optional pickle version. If omitted defaults to 0.

- **shape**: tuple
- **dtype**: data-type
- **isFortran**: bool
- **rawdata**: string or list

A binary string with the data (or a list if ‘a’ is an object array)

**Basic customization:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.<strong>new</strong>()</td>
<td>T.<strong>new</strong>(S, ...) -&gt; a new object with type S, a subtype of T</td>
</tr>
<tr>
<td>ndarray.<strong>array</strong>()</td>
<td>a.<strong>array</strong>(dtype) -&gt; reference if type unchanged, copy otherwise.</td>
</tr>
<tr>
<td>ndarray.<strong>array_wrap</strong>()</td>
<td>a.<strong>array_wrap</strong>(obj) -&gt; Object of same type as ndarray object a.</td>
</tr>
</tbody>
</table>

**__array__()**

```
a.__array__(dtype) -> reference if type unchanged, copy otherwise.
```

Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

**__array_wrap__()**

```
a.__array_wrap__(obj) -> Object of same type as ndarray object a.
```

Container customization: (see Indexing)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.<strong>len</strong>()</td>
<td>x.<strong>getitem</strong>(y) &lt;=&gt; x[y]</td>
</tr>
<tr>
<td>ndarray.<strong>getitem</strong>()</td>
<td>x.<strong>getitem</strong>(i, y) &lt;=&gt; x[i]=y</td>
</tr>
<tr>
<td>ndarray.<strong>setitem</strong>()</td>
<td>x.<strong>setitem</strong>(i, y) &lt;=&gt; x[i]=y</td>
</tr>
<tr>
<td>ndarray.<strong>getslice</strong>()</td>
<td>x.<strong>getslice</strong>(i, j) &lt;=&gt; x[i:j]</td>
</tr>
<tr>
<td>ndarray.<strong>setslice</strong>()</td>
<td>x.<strong>setslice</strong>(i, j, y) &lt;=&gt; x[i:j]=y</td>
</tr>
<tr>
<td>ndarray.<strong>contains</strong>()</td>
<td>x.<strong>contains</strong>(y) &lt;=&gt; y in x</td>
</tr>
</tbody>
</table>

**__len__()**

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

**__getitem__()**

```
x.__getitem__(i, y) <=> x[i]=y
```

**__setitem__()**

```
x.__setitem__(i, y) <=> x[i]=y
```

**__getslice__()**

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

Use of negative indices is not supported.

**__setslice__()**

```
x.__setslice__(i, j, y) <=> x[i:j]=y
```

Use of negative indices is not supported.

**__contains__()**

```
x.__contains__(y) <=> y in x
```

38 Chapter 1. Array objects
Conversion; the operations `complex`, `int`, `long`, `float`, `oct`, and `hex`. They work only on arrays that have one element in them and return the appropriate scalar.

```python
ndarray.__int__() <
ndarray.__long__() <
ndarray.__float__() <
ndarray.__oct__() <
ndarray.__hex__() <
```

__int__()
__long__()
__float__()
__oct__()
__hex__()

String representations:

```python
ndarray.__str__() <
ndarray.__repr__() <
```

__str__()
__repr__()

## 1.2 Scalars

Python defines only one type of a particular data class (there is only one integer type, one floating-point type, etc.). This can be convenient in applications that don’t need to be concerned with all the ways data can be represented in a computer. For scientific computing, however, more control is often needed.

In NumPy, there are 21 new fundamental Python types to describe different types of scalars. These type descriptors are mostly based on the types available in the C language that CPython is written in, with several additional types compatible with Python’s types.

Array scalars have the same attributes and methods as `ndarrays`. This allows one to treat items of an array partly on the same footing as arrays, smoothing out rough edges that result when mixing scalar and array operations.

Array scalars live in a hierarchy (see the Figure below) of data types. They can be detected using the hierarchy: For example, `isinstance(val, np.generic)` will return `True` if `val` is an array scalar object. Alternatively, what kind of array scalar is present can be determined using other members of the data type hierarchy. Thus, for example `isinstance(val, np.complexfloating)` will return `True` if `val` is a complex valued type, while `isinstance(val, np.flexible)` will return true if `val` is one of the flexible itemsize array types (string, unicode, void).

---

1 However, array scalars are immutable, so none of the array scalar attributes are settable.
Figure 1.2: **Figure**: Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.
1.2.1 Built-in scalar types

The built-in scalar types are shown below. Along with their (mostly) C-derived names, the integer, float, and complex data-types are also available using a bit-width convention so that an array of the right size can always be ensured (e.g. int8, float64, complex128). Two aliases (intp and uintp) pointing to the integer type that is sufficiently large to hold a C pointer are also provided. The C-like names are associated with character codes, which are shown in the table. Use of the character codes, however, is discouraged.

Five of the scalar types are essentially equivalent to fundamental Python types and therefore inherit from them as well as from the generic array scalar type:

<table>
<thead>
<tr>
<th>Array scalar type</th>
<th>Related Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_</td>
<td>IntType</td>
</tr>
<tr>
<td>float_</td>
<td>FloatType</td>
</tr>
<tr>
<td>complex_</td>
<td>ComplexType</td>
</tr>
<tr>
<td>str_</td>
<td>StringType</td>
</tr>
<tr>
<td>unicode_</td>
<td>UnicodeType</td>
</tr>
</tbody>
</table>

The bool_ data type is very similar to the Python BooleanType but does not inherit from it because Python’s BooleanType does not allow itself to be inherited from, and on the C-level the size of the actual bool data is not the same as a Python Boolean scalar.

Warning: The bool_ type is not a subclass of the int_ type (the bool_ is not even a number type). This is different than Python’s default implementation of bool as a sub-class of int.

Tip: The default data type in Numpy is float_.

In the tables below, platform? means that the type may not be available on all platforms. Compatibility with different C or Python types is indicated: two types are compatible if their data is of the same size and interpreted in the same way.

Booleans:

<table>
<thead>
<tr>
<th>Type</th>
<th>Remarks</th>
<th>Character code</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool_</td>
<td>compatible: Python bool</td>
<td>'b'</td>
</tr>
<tr>
<td>bool8</td>
<td>8 bits</td>
<td></td>
</tr>
</tbody>
</table>

Integers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Remarks</th>
<th>Character code</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>compatible: C char</td>
<td>'b'</td>
</tr>
<tr>
<td>short</td>
<td>compatible: C short</td>
<td>'h'</td>
</tr>
<tr>
<td>intc</td>
<td>compatible: C int</td>
<td>'i'</td>
</tr>
<tr>
<td>int_</td>
<td>compatible: Python int</td>
<td>'l'</td>
</tr>
<tr>
<td>longlong</td>
<td>compatible: C long long</td>
<td>'q'</td>
</tr>
<tr>
<td>intp</td>
<td>large enough to fit a pointer</td>
<td>'p'</td>
</tr>
<tr>
<td>int8</td>
<td>8 bits</td>
<td></td>
</tr>
<tr>
<td>int16</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>int32</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>int64</td>
<td>64 bits</td>
<td></td>
</tr>
</tbody>
</table>

Unsigned integers:
NumPy Reference, Release 1.4.1

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubyte</td>
<td>compatible: C unsigned char</td>
<td>'B'</td>
</tr>
<tr>
<td>ushort</td>
<td>compatible: C unsigned short</td>
<td>'H'</td>
</tr>
<tr>
<td>uintc</td>
<td>compatible: C unsigned int</td>
<td>'I'</td>
</tr>
<tr>
<td>uint</td>
<td>compatible: Python int</td>
<td>'L'</td>
</tr>
<tr>
<td>ulonglong</td>
<td>compatible: C long long large enough to fit a pointer</td>
<td>'Q'</td>
</tr>
<tr>
<td>uintp</td>
<td>8 bits</td>
<td>'P'</td>
</tr>
<tr>
<td>uint8</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>uint16</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>uint32</td>
<td>64 bits</td>
<td></td>
</tr>
</tbody>
</table>

Floating-point numbers:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>compatible: C float</td>
<td>'f'</td>
</tr>
<tr>
<td>double</td>
<td>compatible: C double</td>
<td>'d'</td>
</tr>
<tr>
<td>float_</td>
<td>compatible: Python float</td>
<td>'g'</td>
</tr>
<tr>
<td>longfloat</td>
<td>compatible: C long float</td>
<td></td>
</tr>
<tr>
<td>float32</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>float64</td>
<td>64 bits</td>
<td></td>
</tr>
<tr>
<td>float96</td>
<td>96 bits, platform?</td>
<td></td>
</tr>
<tr>
<td>float128</td>
<td>128 bits, platform?</td>
<td></td>
</tr>
</tbody>
</table>

Complex floating-point numbers:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>csingle</td>
<td>compatible: Python complex</td>
<td>'F'</td>
</tr>
<tr>
<td>complex_</td>
<td>compatible: Python complex</td>
<td>'D'</td>
</tr>
<tr>
<td>longfloat</td>
<td>two 32-bit floats</td>
<td>'G'</td>
</tr>
<tr>
<td>complex64</td>
<td>two 64-bit floats</td>
<td></td>
</tr>
<tr>
<td>complex128</td>
<td>two 96-bit floats, platform?</td>
<td></td>
</tr>
<tr>
<td>complex192</td>
<td>two 128-bit floats, platform?</td>
<td></td>
</tr>
<tr>
<td>complex256</td>
<td>two 128-bit floats, platform?</td>
<td></td>
</tr>
</tbody>
</table>

Any Python object:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_</td>
<td>any Python object</td>
<td>'O'</td>
</tr>
</tbody>
</table>

Note: The data actually stored in object arrays (i.e., arrays having dtype object_) are references to Python objects, not the objects themselves. Hence, object arrays behave more like usual Python lists, in the sense that their contents need not be of the same Python type.

The object type is also special because an array containing object_ items does not return an object_ object on item access, but instead returns the actual object that the array item refers to.

The following data types are flexible. They have no predefined size: the data they describe can be of different length in different arrays. (In the character codes # is an integer denoting how many elements the data type consists of.)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>str_</td>
<td>compatible: Python str</td>
<td>'S#'</td>
</tr>
<tr>
<td>unicode_</td>
<td>compatible: Python unicode</td>
<td>'U#'</td>
</tr>
<tr>
<td>void</td>
<td></td>
<td>'V#'</td>
</tr>
</tbody>
</table>

Warning: Numeric Compatibility: If you used old typecode characters in your Numeric code (which was never recommended), you will need to change some of them to the new characters. In particular, the needed changes are c -> S1, b -> B, l -> h, s -> h, w -> H, and u -> I. These changes make the type character convention more consistent with other Python modules such as the struct module.
1.2.2 Attributes

The array scalar objects have an array priority of NPY_SCALAR_PRIORITY (-1,000,000.0). They also do not (yet) have a ctypes attribute. Otherwise, they share the same attributes as arrays:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic.flags</td>
<td>integer value of flags</td>
</tr>
<tr>
<td>generic.shape</td>
<td>tuple of array dimensions</td>
</tr>
<tr>
<td>generic.strides</td>
<td>tuple of bytes steps in each dimension</td>
</tr>
<tr>
<td>generic.ndim</td>
<td>number of array dimensions</td>
</tr>
<tr>
<td>generic.data</td>
<td>pointer to start of data</td>
</tr>
<tr>
<td>generic.size</td>
<td>number of elements in the gentype</td>
</tr>
<tr>
<td>generic.itemsize</td>
<td>length of one element in bytes</td>
</tr>
<tr>
<td>generic.base</td>
<td>base object</td>
</tr>
<tr>
<td>generic.dtype</td>
<td>get array data-descriptor</td>
</tr>
<tr>
<td>generic.real</td>
<td>real part of scalar</td>
</tr>
<tr>
<td>generic.imag</td>
<td>imaginary part of scalar</td>
</tr>
<tr>
<td>generic.flat</td>
<td>a 1-d view of scalar</td>
</tr>
<tr>
<td>generic.T</td>
<td>transpose</td>
</tr>
<tr>
<td><strong>array_interface</strong></td>
<td>Array protocol: Python side</td>
</tr>
<tr>
<td><strong>array_struct</strong></td>
<td>Array protocol: struct</td>
</tr>
<tr>
<td><strong>array_priority</strong></td>
<td>Array priority.</td>
</tr>
<tr>
<td><strong>array_wrap</strong>(obj)</td>
<td>sc.<strong>array_wrap</strong>(obj) return scalar from array</td>
</tr>
</tbody>
</table>

flags
integer value of flags

shape
tuple of array dimensions

strides
tuple of bytes steps in each dimension

ndim
number of array dimensions

data
pointer to start of data

size
number of elements in the gentype

itemsize
length of one element in bytes

base
base object

dtype
get array data-descriptor

real
real part of scalar

imag
imaginary part of scalar

flat
a 1-d view of scalar

T
transpose
__array_interface__

Array protocol: Python side

__array_struct__

Array protocol: struct

__array_priority__

Array priority.

__array_wrap__()

sc.__array_wrap__(obj) return scalar from array

1.2.3 Indexing

See Also:

Indexing, Data type objects (dtype)

Array scalars can be indexed like 0-dimensional arrays: if \( x \) is an array scalar,

- \( x[()] \) returns a 0-dimensional ndarray
- \( x[\text{'field-name'}] \) returns the array scalar in the field field-name. (\( x \) can have fields, for example, when it corresponds to a record data type.)

1.2.4 Methods

Array scalars have exactly the same methods as arrays. The default behavior of these methods is to internally convert the scalar to an equivalent 0-dimensional array and to call the corresponding array method. In addition, math operations on array scalars are defined so that the same hardware flags are set and used to interpret the results as for ufunc, so that the error state used for ufuncs also carries over to the math on array scalars.

The exceptions to the above rules are given below:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic.<strong>array</strong></td>
<td>Base class for numpy scalar types.</td>
</tr>
<tr>
<td>generic.<strong>array</strong>(type)</td>
<td>sc.<strong>array</strong>(type) return 0-dim array</td>
</tr>
<tr>
<td>generic.<strong>array_wrap</strong></td>
<td>sc.<strong>array_wrap</strong>(obj) return scalar from array</td>
</tr>
<tr>
<td>generic.<strong>squeeze</strong></td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>generic.byteswap()</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>generic.<strong>reduce</strong></td>
<td>Class from which most (all?) numpy scalar types are derived. For consistency, exposes the same API as ndarray, despite many consequent attributes being either “get-only,” or completely irrelevant. This is the class from which it is strongly suggested users should derive custom scalar types.</td>
</tr>
<tr>
<td>generic.<strong>setstate</strong></td>
<td>Methods</td>
</tr>
<tr>
<td><strong>array</strong>(type)</td>
<td>sc.<strong>array</strong>(type) return 0-dim array</td>
</tr>
<tr>
<td><strong>array_wrap</strong>(obj)</td>
<td>sc.<strong>array_wrap</strong>(obj) return scalar from array</td>
</tr>
</tbody>
</table>
byteswap()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
See Also:
The
__reduce__()

__setstate__()

setflags()
Not implemented (virtual attribute)

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

1.2.5 Defining new types

There are two ways to effectively define a new array scalar type (apart from composing record dtypes from the built-in scalar types): One way is to simply subclass the ndarray and overwrite the methods of interest. This will work to a degree, but internally certain behaviors are fixed by the data type of the array. To fully customize the data type of an array you need to define a new data-type, and register it with NumPy. Such new types can only be defined in C, using the Numpy C-API.

1.3 Data type objects (dtype)

A data type object (an instance of numpy.dtype class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:

1. Type of the data (integer, float, Python object, etc.)
2. Size of the data (how many bytes is in e.g. the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is a record, an aggregate of other data types, (e.g., describing an array item consisting of an integer and a float),
   (a) what are the names of the “fields” of the record, by which they can be accessed,
   (b) what is the data-type of each field, and
   (c) which part of the memory block each field takes.
5. If the data is a sub-array, what is its shape and data type.

To describe the type of scalar data, there are several built-in scalar types in Numpy for various precision of integers, floating-point numbers, etc. An item extracted from an array, e.g., by indexing, will be a Python object whose type is the scalar type associated with the data type of the array.
Note that the scalar types are not `dtype` objects, even though they can be used in place of one whenever a data type specification is needed in Numpy. Record data types are formed by creating a data type whose `fields` contain other data types. Each field has a name by which it can be `accessed`. The parent data type should be of sufficient size to contain all its fields; the parent can for example be based on the `void` type which allows an arbitrary item size. Record data types may also contain other record types and fixed-size sub-array data types in their fields. Finally, a data type can describe items that are themselves arrays of items of another data type. These sub-arrays must, however, be of a fixed size. If an array is created using a data-type describing a sub-array, the dimensions of the sub-array are appended to the shape of the array when the array is created. Sub-arrays in a field of a record behave differently, see `Record Access`.

**Example**

A simple data type containing a 32-bit big-endian integer: (see *Specifying and constructing data types* for details on construction)

```
>>> dt = np.dtype('>i4')
>>> dt.byteorder
>'
>>> dt.itemsize
4
>>> dt.name
'int32'
>>> dt.type is np.int32
True
```

The corresponding array scalar type is `int32`.

**Example**

A record data type containing a 16-character string (in field ‘name’) and a sub-array of two 64-bit floating-point number (in field ‘grades’):

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt['name']
dtype('|S16')
>>> dt['grades']
dtype(('float64',(2,)))
```

Items of an array of this data type are wrapped in an *array scalar* type that also has two fields:

```
>>> x = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
>>> x[1]
('John', [6.0, 7.0])
>>> x[1]['grades']
array([ 6., 7.])
>>> type(x[1])
<type 'numpy.void'>
>>> type(x[1]['grades'])
<type 'numpy.ndarray'>
```

### 1.3.1 Specifying and constructing data types

Whenever a data-type is required in a NumPy function or method, either a `dtype` object or something that can be converted to one can be supplied. Such conversions are done by the `dtype` constructor:

```
dtype(obj[, align, copy])  Create a data type object.
```
class dtype()

Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

Parameters

- **obj**: Object to be converted to a data type object.
- **align**: bool, optional
  
  Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be True only if obj is a dictionary or a comma-separated string.

- **copy**: bool, optional
  
  Make a new copy of the data-type object. If False, the result may just be a reference to a built-in data-type object.

Examples

Using array-scalar type:

```python
>>> np.dtype(np.int16)
dtype('int16')
```

Record, one field name ‘f1’, containing int16:

```python
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Record, one field named ‘f1’, in itself containing a record with one field:

```python
>>> np.dtype([('f1', [{'f1': np.int16}])])
dtype([('f1', [('f1', '<i2')])])
```

Record, two fields: the first field contains an unsigned int, the second an int32:

```python
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```

Using array-protocol type strings:

```python
>>> np.dtype([('a','f8'),('b','S10')])
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```python
>>> np.dtype("i4, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3)]])
```

Using tuples. int is a fixed type, 3 the field’s shape. void is a flexible type, here of size 10:

```python
>>> np.dtype([('hello',(np.int,3)),('world',np.void,10)])
dtype([('hello', '<i4', 3), ('world', '|V10')])
```

Subdivide int16 into 2 int8’s, called x and y. 0 and 1 are the offsets in bytes:
NumPy Reference, Release 1.4.1

>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype(('i2', [('x', '|i1'), ('y', '|i1')]))

Using dictionaries. Two fields named 'gender' and 'age':

>>> np.dtype({‘names’:[‘gender’,’age’], ’formats’:[’S1’,np.uint8]})
dtype([('gender', '|S1'), ('age', '|u1')])

Offsets in bytes, here 0 and 25:

>>> np.dtype({‘surname’:(‘S25’,0),’age’:(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])

Methods

What can be converted to a data-type object is described below:

dtype object

Used as-is.

None

The default data type: float_.

Array-scalar types

The 21 built-in array scalar type objects all convert to an associated data-type object. This is true for their sub-classes as well.

Note that not all data-type information can be supplied with a type-object: for example, flexible data-types have a default itemsize of 0, and require an explicitly given size to be useful.

Example

>>> dt = np.dtype(np.int32)       # 32-bit integer
>>> dt = np.dtype(np.complex128)  # 128-bit complex floating-point number

Generic types

The generic hierarchical type objects convert to corresponding type objects according to the associations:

<table>
<thead>
<tr>
<th>number, inexact, floating</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>complexfloating</td>
<td>cfloat</td>
</tr>
<tr>
<td>integer, signedinteger</td>
<td>int_</td>
</tr>
<tr>
<td>unsignedinteger</td>
<td>uint</td>
</tr>
<tr>
<td>character</td>
<td>string</td>
</tr>
<tr>
<td>generic, flexible</td>
<td>void</td>
</tr>
</tbody>
</table>

Built-in Python types

Several python types are equivalent to a corresponding array scalar when used to generate a dtype object:
Example

```python
>>> dt = np.dtype(float)  # Python-compatible floating-point number
>>> dt = np.dtype(int)   # Python-compatible integer
>>> dt = np.dtype(object)  # Python object
```

Types with `.dtype`

Any type object with a `.dtype` attribute: The attribute will be accessed and used directly. The attribute must return something that is convertible into a dtype object.

Several kinds of strings can be converted. Recognized strings can be prepended with `'>` (big-endian), `'<` (little-endian), or `'=` (hardware-native, the default), to specify the byte order.

One-character strings

Each built-in data-type has a character code (the updated Numeric typecodes), that uniquely identifies it.

Example

```python
>>> dt = np.dtype('b')  # byte, native byte order
>>> dt = np.dtype('>H')  # big-endian unsigned short
>>> dt = np.dtype('<f')  # little-endian single-precision float
>>> dt = np.dtype('d')  # double-precision floating-point number
```

Array-protocol type strings (see The Array Interface)

The first character specifies the kind of data and the remaining characters specify how many bytes of data. The supported kinds are

<table>
<thead>
<tr>
<th>Character code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>Boolean</td>
</tr>
<tr>
<td>'i'</td>
<td>(signed) integer</td>
</tr>
<tr>
<td>'u'</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>'f'</td>
<td>floating-point</td>
</tr>
<tr>
<td>'c'</td>
<td>complex-floating point</td>
</tr>
<tr>
<td>'S', 'a'</td>
<td>string</td>
</tr>
<tr>
<td>'U'</td>
<td>unicode</td>
</tr>
<tr>
<td>'V'</td>
<td>anything (void)</td>
</tr>
</tbody>
</table>

Example

```python
>>> dt = np.dtype('i4')  # 32-bit signed integer
>>> dt = np.dtype('f8')  # 64-bit floating-point number
>>> dt = np.dtype('c16')  # 128-bit complex floating-point number
>>> dt = np.dtype('a25')  # 25-character string
```

String with comma-separated fields

Numarray introduced a short-hand notation for specifying the format of a record as a comma-separated string of basic formats.
A basic format in this context is an optional shape specifier followed by an array-protocol type string. Parenthesis are required on the shape if it is greater than 1-d. NumPy allows a modification on the format in that any string that can uniquely identify the type can be used to specify the data-type in a field. The generated data-type fields are named ‘f0’, ‘f2’, ..., ‘f<N-1>’ where N (>1) is the number of comma-separated basic formats in the string. If the optional shape specifier is provided, then the data-type for the corresponding field describes a sub-array.

**Example**

- field named f0 containing a 32-bit integer
- field named f1 containing a 2 x 3 sub-array of 64-bit floating-point numbers
- field named f2 containing a 32-bit floating-point number

```python
>>> dt = np.dtype("i4, (2,3)f8, f4")
```

- field named f0 containing a 3-character string
- field named f1 containing a sub-array of shape (3,) containing 64-bit unsigned integers
- field named f2 containing a 3 x 4 sub-array containing 10-character strings

```python
>>> dt = np.dtype("a3, 3u8, (3,4)a10")
```

**Type strings**

Any string in `numpy.sctypeDict.keys()`:

**Example**

```python
>>> dt = np.dtype('uint32')    # 32-bit unsigned integer
>>> dt = np.dtype('Float64')   # 64-bit floating-point number
```

(flexible_dtype, itemsize)

The first argument must be an object that is converted to a flexible data-type object (one whose element size is 0), the second argument is an integer providing the desired itemsize.

**Example**

```python
>>> dt = np.dtype((void, 10)) # 10-byte wide data block
>>> dt = np.dtype((str, 35))  # 35-character string
>>> dt = np.dtype(('U', 10))  # 10-character unicode string
```

(fixed_dtype, shape)

The first argument is any object that can be converted into a fixed-size data-type object. The second argument is the desired shape of this type. If the shape parameter is 1, then the data-type object is equivalent to fixed dtype. If `shape` is a tuple, then the new dtype defines a sub-array of the given shape.

**Example**

```python
>>> dt = np.dtype((np.int32, (2,2)))  # 2 x 2 integer sub-array
>>> dt = np.dtype(('S10', 1))         # 10-character string
>>> dt = np.dtype(("i4, (2,3)f8, f4", (2,3))) # 2 x 3 record sub-array
```

(base_dtype, new_dtype)
Both arguments must be convertible to data-type objects in this case. The base_dtype is the data-type object that the new data-type builds on. This is how you could assign named fields to any built-in data-type object.

**Example**

32-bit integer, whose first two bytes are interpreted as an integer via field real, and the following two bytes via field imag.

```python
>>> dt = np.dtype((np.int32, {'real': (np.int16, 0), 'imag': (np.int16, 2)})
```

32-bit integer, which is interpreted as consisting of a sub-array of shape (4,) containing 8-bit integers:

```python
>>> dt = np.dtype((np.int32, (np.int8, 4)))
```

32-bit integer, containing fields r, g, b, a that interpret the 4 bytes in the integer as four unsigned integers:

```python
>>> dt = np.dtype(('i4', [('r','u1'), ('g','u1'), ('b','u1'), ('a','u1')])
```

**Note:** XXX: does the second-to-last example above make sense? [[field_name, field_dtype, field_shape], ...]

`obj` should be a list of fields where each field is described by a tuple of length 2 or 3. (Equivalent to the descr item in the __array_interface__ attribute.)

The first element, field_name, is the field name (if this is " then a standard field name, 'f#', is assigned). The field name may also be a 2-tuple of strings where the first string is either a “title” (which may be any string or unicode string) or meta-data for the field which can be any object, and the second string is the “name” which must be a valid Python identifier.

The second element, field_dtype, can be anything that can be interpreted as a data-type.

The optional third element field_shape contains the shape if this field represents an array of the data-type in the second element. Note that a 3-tuple with a third argument equal to 1 is equivalent to a 2-tuple.

This style does not accept align in the dtype constructor as it is assumed that all of the memory is accounted for by the array interface description.

**Example**

Data-type with fields big (big-endian 32-bit integer) and little (little-endian 32-bit integer):

```python
>>> dt = np.dtype([('big', '>i4'), ('little', '<i4')])
```

Data-type with fields R, G, B, A, each being an unsigned 8-bit integer:

```python
>>> dt = np.dtype(('R', 'u1'), ('G', 'u1'), ('B', 'u1'), ('A', 'u1'))
```

`{\'names\': ..., \'formats\': ..., \'offsets\': ..., \'titles\': ...}`

This style has two required and two optional keys. The names and formats keys are required. Their respective values are equal-length lists with the field names and the field formats. The field names must be strings and the field formats can be any object accepted by dtype constructor.

The optional keys in the dictionary are offsets and titles and their values must each be lists of the same length as the names and formats lists. The offsets value is a list of byte offsets (integers) for each field, while the titles value is a list of titles for each field (None can be used if no title is desired for that field). The titles can be any string or unicode object and will add another entry to the fields dictionary keyed by the title and referencing the same field tuple which will contain the title as an additional tuple member.
Example

Data type with fields \( r, g, b, a \), each being a 8-bit unsigned integer:

```python
>>> dt = np.dtype({'names': ['r', 'g', 'b', 'a'],
                 'formats': [uint8, uint8, uint8, uint8]})
```

Data type with fields \( r \) and \( b \) (with the given titles), both being 8-bit unsigned integers, the first at byte position 0 from the start of the field and the second at position 2:

```python
>>> dt = np.dtype({'names': ['r', 'b'], 'formats': ['u1', 'u1'],
                 'offsets': [0, 2],
                 'titles': ['Red pixel', 'Blue pixel']})
```

('field1': ..., 'field2': ..., ...)

This style allows passing in the `fields` attribute of a data-type object.

`obj` should contain string or unicode keys that refer to `(data-type, offset)` or `(data-type, offset, title)` tuples.

Example

Data type containing field `col1` (10-character string at byte position 0), `col2` (32-bit float at byte position 10), and `col3` (integers at byte position 14):

```python
>>> dt = np.dtype({'col1': ('S10', 0), 'col2': (float32, 10), 'col3': (int, 14)})
```

### 1.3.2 `dtype`

Numpy data type descriptions are instances of the `dtype` class.

**Attributes**

The type of the data is described by the following `dtype` attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtype.type</code></td>
<td>The type object used to instantiate a scalar of this data-type.</td>
</tr>
<tr>
<td><code>dtype.kind</code></td>
<td>A character code (one of ‘biufcSUV’) identifying the general kind of data.</td>
</tr>
<tr>
<td><code>dtype.char</code></td>
<td>A unique character code for each of the 21 different built-in types.</td>
</tr>
<tr>
<td><code>dtype.num</code></td>
<td>A unique number for each of the 21 different built-in types.</td>
</tr>
<tr>
<td><code>dtype.str</code></td>
<td>The array-protocol typestring of this data-type object.</td>
</tr>
</tbody>
</table>

`type`  
The type object used to instantiate a scalar of this data-type.

`kind`  
A character code (one of ‘biufcSUV’) identifying the general kind of data.

`char`  
A unique character code for each of the 21 different built-in types.

`num`  
A unique number for each of the 21 different built-in types.

These are roughly ordered from least-to-most precision.

`str`  
The array-protocol typestring of this data-type object.
Size of the data is in turn described by:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtype.name</code></td>
<td>A bit-width name for this data-type.</td>
</tr>
<tr>
<td><code>dtype.itemsize</code></td>
<td>The element size of this data-type object.</td>
</tr>
</tbody>
</table>

#### name
A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.

#### itemsize
The element size of this data-type object.

For 18 of the 21 types this number is fixed by the data-type. For the flexible data-types, this number can be anything.

#### Endianness of this data:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtype.byteorder</code></td>
<td>A character indicating the byte-order of this data-type object.</td>
</tr>
</tbody>
</table>

#### byteorder
A character indicating the byte-order of this data-type object.

One of:

- `'='` native
- `'<'` little-endian
- `'>` big-endian
- `'|'` not applicable

All built-in data-type objects have byteorder either `=' or `'|'.

**Examples**

```python
>>> dt = np.dtype('i2')
>>> dt.byteorder
'='
>>> # endian is not relevant for 8 bit numbers
>>> np.dtype('i1').byteorder
'|'
>>> # or ASCII strings
>>> np.dtype('S2').byteorder
'|'
>>> # Even if specific code is given, and it is native
>>> # '=' is the byteorder
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>
>>> swapped_code = sys_is_le and '>' or '<'
>>> dt = np.dtype(native_code + 'i2')
>>> dt.byteorder
'='
>>> # Swapped code shows up as itself
>>> dt = np.dtype(swapped_code + 'i2')
>>> dt.byteorder == swapped_code
True
```

Information about sub-data-types in a *record*:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtype.fields</code></td>
<td>Dictionary of named fields defined for this data type, or None.</td>
</tr>
<tr>
<td><code>dtype.names</code></td>
<td>Ordered list of field names, or None if there are no fields.</td>
</tr>
</tbody>
</table>

1.3. Data type objects (*dtype*)
fields

Dictionary of named fields defined for this data type, or None.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

\[
\text{(dtype, offset[, title])}
\]

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the `ndarray.getfield` and `ndarray.setfield` methods.

See Also:

`ndarray.getfield`, `ndarray.setfield`

Examples

```python
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print dt.fields
{'grades': (dtype(('float64',(2,))), 16), 'name': (dtype('|S16'), 0)}
```

names

Ordered list of field names, or None if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

Examples

```python
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print dt.names
('name', 'grades')
```

For data types that describe sub-arrays:

<table>
<thead>
<tr>
<th>dtype.subdtype</th>
<th>Tuple (item_dtype, shape) if this dtype describes a sub-array, and None otherwise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype.shape</td>
<td>Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.</td>
</tr>
</tbody>
</table>

subdtype

Tuple (item_dtype, shape) if this dtype describes a sub-array, and None otherwise.

The shape is the fixed shape of the sub-array described by this data type, and item_dtype the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by shape are tacked on to the end of the retrieved array.

shape

Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.

Attributes providing additional information:

<table>
<thead>
<tr>
<th>dtype.hasobject</th>
<th>Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype.flags</td>
<td>Bit-flags describing how this data type is to be interpreted.</td>
</tr>
<tr>
<td>dtype.isbuiltin</td>
<td>Integer indicating how this dtype relates to the built-in dtypes.</td>
</tr>
<tr>
<td>dtype.isnative</td>
<td>Boolean indicating whether the byte order of this dtype is native to the platform.</td>
</tr>
<tr>
<td>dtype.descr</td>
<td>Array-interface compliant full description of the data-type.</td>
</tr>
<tr>
<td>dtype.alignment</td>
<td>The required alignment (bytes) of this data-type according to the compiler.</td>
</tr>
</tbody>
</table>
**hasobject**

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won’t.

**flags**

Bit-flags describing how this data type is to be interpreted.

Bit-masks are in `numpy.core.multiarray` as the constants `ITEM_HASOBJECT`, `LIST_PICKLE`, `ITEM_IS_POINTER`, `NEEDS_INIT`, `NEEDS_PYAPI`, `USE_GETITEM`, `USE_SETITEM`. A full explanation of these flags is in C-API documentation; they are largely useful for user-defined data-types.

**isbuiltin**

Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if this is a structured array type, with fields</td>
</tr>
<tr>
<td>1</td>
<td>if this is a dtype compiled into numpy (such as ints, floats etc)</td>
</tr>
<tr>
<td>2</td>
<td>if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See User-defined data-types (in NumPy User Guide) in the Numpy manual.</td>
</tr>
</tbody>
</table>

**Examples**

```python
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

**isnative**

Boolean indicating whether the byte order of this dtype is native to the platform.

**descr**

Array-interface compliant full description of the data-type.

The format is that required by the ‘descr’ key in the `__array_interface__` attribute.

**alignment**

The required alignment (bytes) of this data-type according to the compiler.

More information is available in the C-API section of the manual.

**Methods**

Data types have the following method for changing the byte order:

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

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

- **new_order**: string, optional
  
  Byte order to force; a value from the byte order specifications below. The default value (`'S'`) results in swapping the current byte order. *new_order* codes can be any of:

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

  The code does a case-insensitive check on the first letter of *new_order* for these alternatives. For example, any of `'>` or `B` or `b` or `brian` are valid to specify big-endian.

Returns

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

Notes

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

Examples

```python
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>
>>> swapped_code = sys_is_le and '>' or '<
>>> native_dt = np.dtype(native_code+'i2'))
>>> swapped_dt = np.dtype(swapped_code+'i2'))
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

The following methods implement the pickle protocol:

```
dtype.__reduce__()
dtype.__setstate__()
__reduce__()
```
1.4 Indexing

ndarrays can be indexed using the standard Python \texttt{x[obj]} syntax, where \texttt{x} is the array and \texttt{obj} the selection. There are three kinds of indexing available: record access, basic slicing, advanced indexing. Which one occurs depends on \texttt{obj}.

\textbf{Note:} In Python, \texttt{x[(exp1, exp2, ..., expN)]} is equivalent to \texttt{x[exp1, exp2, ..., expN]}; the latter is just syntactic sugar for the former.

1.4.1 Basic Slicing

Basic slicing extends Python’s basic concept of slicing to N dimensions. Basic slicing occurs when \texttt{obj} is a slice object (constructed by start:stop:step notation inside of brackets), an integer, or a tuple of slice objects and integers. Ellipsis and newaxis objects can be interspersed with these as well. In order to remain backward compatible with a common usage in Numeric, basic slicing is also initiated if the selection object is any sequence (such as a list) containing slice objects, the Ellipsis object, or the newaxis object, but no integer arrays or other embedded sequences. The simplest case of indexing with \texttt{N} integers returns an array scalar representing the corresponding item. As in Python, all indices are zero-based: for the \texttt{i}-th index \texttt{n}, the valid range is \(0 \leq n_i < d_i\) where \(d_i\) is the \(i\)-th element of the shape of the array. Negative indices are interpreted as counting from the end of the array (i.e., if \(i < 0\), it means \(n_i + i\)).

All arrays generated by basic slicing are always views of the original array.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is \texttt{i:j:k} where \texttt{i} is the starting index, \texttt{j} is the stopping index, and \texttt{k} is the step (\(k \neq 0\)). This selects the \texttt{m} elements (in the corresponding dimension) with index values \(i, i + k, ..., i + (m - 1) k\) where \(m = q + (r \neq 0)\) and \(q\) and \(r\) are the quotient and remainder obtained by dividing \(j - i\) by \(k\): \(j - i = q k + r\), so that \(i + (m - 1) k < j\).

\textbf{Example}

\begin{verbatim}
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[1:7:2]
array([1, 3, 5])
\end{verbatim}

- Negative \texttt{i} and \texttt{j} are interpreted as \texttt{n + i} and \texttt{n + j} where \texttt{n} is the number of elements in the corresponding dimension. Negative \texttt{k} makes stepping go towards smaller indices.

\textbf{Example}

\begin{verbatim}
>>> x[-2:10]
array([8, 9])
>>> x[-3:3:-1]
array([7, 6, 5, 4])
\end{verbatim}

- Assume \texttt{n} is the number of elements in the dimension being sliced. Then, if \texttt{i} is not given it defaults to \texttt{0} for \texttt{k > 0} and \texttt{n} for \texttt{k < 0}. If \texttt{j} is not given it defaults to \texttt{n} for \texttt{k > 0} and \texttt{-1} for \texttt{k < 0}. If \texttt{k} is not given it defaults to \texttt{1}.

\textbf{Note:} \texttt{::} is the same as \texttt{:} and means select all indices along this axis.

\textbf{Example}

\begin{verbatim}
...
>>> x[5:]
array([5, 6, 7, 8, 9])

- If the number of objects in the selection tuple is less than \( N \), then \( : \) is assumed for any subsequent dimensions.

**Example**

```python
>>> x = np.array([[[1],[2],[3]], [[4],[5],[6]]])
>>> x.shape
(2, 3, 1)
>>> x[1:2]
array([[4],
       [5],
       [6]])
```

- **Ellipsis** expand to the number of \( : \) objects needed to make a selection tuple of the same length as \( x.ndim \). Only the first ellipsis is expanded, any others are interpreted as \( : \).

**Example**

```python
>>> x[...]#0
array([[1, 2, 3],
       [4, 5, 6]])
```

- Each **newaxis** object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the **newaxis** object in the selection tuple.

**Example**

```python
>>> x[:, np.newaxis, :, :].shape
(2, 1, 3, 1)
```

- An integer, \( i \), returns the same values as \( i:i+1 \) except the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the \( p \)-th element an integer (and all other entries \( : \)) returns the corresponding sub-array with dimension \( N - I \). If \( N = I \) then the returned object is an array scalar. These objects are explained in **Scalars**.

- If the selection tuple has all entries : except the \( p \)-th entry which is a slice object \( i:j:k \), then the returned array has dimension \( N \) formed by concatenating the sub-arrays returned by integer indexing of elements \( i, i+k, \ldots, i + (m - 1) k < j \).

- Basic slicing with more than one non-: entry in the slicing tuple, acts like repeated application of slicing using a single non-: entry, where the non-: entries are successively taken (with all other non-: entries replaced by :). Thus, \( x[ind1,...,ind2,:]: \) acts like \( x[ind1][...,ind2,:]: \) under basic slicing.

**Warning:** The above is **not** true for advanced slicing.

- You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in \( x[obj] = \text{value} \) must be (broadcastable) to the same shape as \( x[obj] \).

**Note:** Remember that a slicing tuple can always be constructed as **obj** and used in the \( x[obj] \) notation. Slice objects can be used in the construction in place of the \( [\text{start:stop:step}] \) notation. For example, \( x[1:10:5,:,:-1] \) can also be implemented as \( \text{obj} = (\text{slice}(1,10,5), \text{slice}(None,None,-1)) \); \( x[\text{obj}] \). This can be useful for constructing generic code that works on arrays of arbitrary dimension.
The `newaxis` object can be used in the basic slicing syntax discussed above. `None` can also be used instead of `newaxis`.

### 1.4.2 Advanced indexing

Advanced indexing is triggered when the selection object, `obj`, is a non-tuple sequence object, an `ndarray` (of data type integer or bool), or a tuple with at least one sequence object or `ndarray` (of data type integer or bool). There are two types of advanced indexing: integer and Boolean.

Advanced indexing always returns a *copy* of the data (contrast with basic slicing that returns a *view*).

#### Integer

Integer indexing allows selection of arbitrary items in the array based on their $N$-dimensional index. This kind of selection occurs when advanced indexing is triggered and the selection object is not an array of data type bool. For the discussion below, when the selection object is not a tuple, it will be referred to as if it had been promoted to a 1-tuple, which will be called the selection tuple. The rules of advanced integer-style indexing are:

- If the length of the selection tuple is larger than $N$ an error is raised.
- All sequences and scalars in the selection tuple are converted to `intp` indexing arrays.
- All selection tuple objects must be convertible to `intp` arrays, `slice` objects, or the `Ellipsis` object.
- The first `Ellipsis` object will be expanded, and any other `Ellipsis` objects will be treated as full slice (:) objects. The expanded `Ellipsis` object is replaced with as many full slice (:) objects as needed to make the length of the selection tuple $N$.
- If the selection tuple is smaller than $N$, then as many : objects as needed are added to the end of the selection tuple so that the modified selection tuple has length $N$.
- All the integer indexing arrays must be *broadcastable* to the same shape.
- The shape of the output (or the needed shape of the object to be used for setting) is the broadcasted shape.
- After expanding any ellipses and filling out any missing : objects in the selection tuple, then let $N_i$ be the number of indexing arrays, and let $N_s = N - N_i$ be the number of slice objects. Note that $N_i > 0$ (or we wouldn’t be doing advanced integer indexing).
- If $N_s = 0$ then the $M$-dimensional result is constructed by varying the index tuple $(i_1, \ldots, i_M)$ over the range of the result shape and for each value of the index tuple $(\text{ind}_1, \ldots, \text{ind}_M)$:

  ```
  \text{result}[i_1, \ldots, i_M] \equiv \text{x[\text{ind}_1[i_1, \ldots, i_M], \text{ind}_2[i_1, \ldots, i_M],}
  \ldots, \text{ind}_N[i_1, \ldots, i_M]]
  ```

- If $N_s > 0$, then partial indexing is done. This can be somewhat mind-boggling to understand, but if you think in terms of the shapes of the arrays involved, it can be easier to grasp what happens. In simple cases (*i.e.* one indexing array and $N - 1$ slice objects) it does exactly what you would expect (concatenation of repeated application of basic slicing). The rule for partial indexing is that the shape of the result (or the interpreted shape of the object to be used in setting) is the shape of $x$ with the indexed subspace replaced with the broadcasted
indexing subspace. If the index subspaces are right next to each other, then the broadcasted indexing space directly replaces all of the indexed subspaces in \( x \). If the indexing subspaces are separated (by slice objects), then the broadcasted indexing space is first, followed by the sliced subspace of \( x \).

**Example**

Suppose \( x.shape \) is \((10,20,30)\) and \( \text{ind} \) is a \((2,3,4)\)-shaped indexing \( \text{intp} \) array, then \( \text{result} = x[\ldots, \text{ind}, :] \) has shape \((10,2,3,4,30)\) because the \((20,)\)-shaped subspace has been replaced with a \((2,3,4)\)-shaped broadcasted indexing subspace. If we let \( i, j, k \) loop over the \((2,3,4)\)-shaped subspace then \( \text{result}[\ldots, i, j, k, : ] = x[\ldots, \text{ind}[i, j, k], :] \). This example produces the same result as \( x.take(\text{ind}, axis=-2) \).

**Example**

Now let \( x.shape \) be \((10,20,30,40,50)\) and suppose \( \text{ind}_1 \) and \( \text{ind}_2 \) are broadcastable to the shape \((2,3,4)\). Then \( x[:, \text{ind}_1, \text{ind}_2] \) has shape \((10,2,3,4,40,50)\) because the \((20,30)\)-shaped subspace from \( X \) has been replaced with the \((2,3,4)\) subspace from the indices. However, \( x[:, \text{ind}_1, :, \text{ind}_2] \) has shape \((2,3,4,10,30,50)\) because there is no unambiguous place to drop in the indexing subspace, thus it is tacked-on to the beginning. It is always possible to use \( \text{.transpose()} \) to move the subspace anywhere desired. (Note that this example cannot be replicated using \( \text{take} \).)

**Boolean**

This advanced indexing occurs when \( \text{obj} \) is an array object of Boolean type (such as may be returned from comparison operators). It is always equivalent to (but faster than) \( x[\text{obj}.\text{nonzero()}] \) where, as described above, \( \text{obj}.\text{nonzero()} \) returns a tuple (of length \( \text{obj.ndim} \)) of integer index arrays showing the *True* elements of \( \text{obj} \).

The special case when \( \text{obj.ndim} == x.\text{ndim} \) is worth mentioning. In this case \( x[\text{obj}] \) returns a 1-dimensional array filled with the elements of \( x \) corresponding to the *True* values of \( \text{obj} \). The search order will be C-style (last index varies the fastest). If \( \text{obj} \) has *True* values at entries that are outside of the bounds of \( x \), then an index error will be raised.

You can also use Boolean arrays as element of the selection tuple. In such instances, they will always be interpreted as \( \text{nonzero(\text{obj})} \) and the equivalent integer indexing will be done.

| **Warning:** | The definition of advanced indexing means that \( x[(1,2,3),] \) is fundamentally different than \( x[(1,2,3)] \). The latter is equivalent to \( x[1,2,3] \) which will trigger basic selection while the former will trigger advanced indexing. Be sure to understand why this is occurs. Also recognize that \( x[[1,2,3]] \) will trigger advanced indexing, whereas \( x[[1,2,\text{slice(None)}]] \) will trigger basic slicing. |

**Note:** XXX: this section may need some tuning... Also the above warning needs explanation as the last part is at odds with the definition of basic indexing.

### 1.4.3 Record Access

**See Also:**

*Data type objects (dtype), Scalars*

If the \( \text{ndarray} \) object is a record array, i.e. its data type is a *record* data type, the *fields* of the array can be accessed by indexing the array with strings, dictionary-like.

Indexing \( x[\text{‘field-name’} \}] \) returns a new view to the array, which is of the same shape as \( x \) (except when the field is a sub-array) but of data type \( x.dtype[\text{‘field-name’}] \) and contains only the part of the data in the specified field. Also record array scalars can be “indexed” this way.
If the accessed field is a sub-array, the dimensions of the sub-array are appended to the shape of the result.

**Example**

```python
g  >>> x = np.zeros((2,2), dtype=[('a', np.int32), ('b', np.float64, (3,3))])
  >>> x['a'].shape
  (2, 2)
  >>> x['a'].dtype
dtype('int32')
  >>> x['b'].shape
  (2, 2, 3, 3)
  >>> x['b'].dtype
dtype('float64')
```

1.4.4 Flat Iterator indexing

The `x.flat` returns an iterator that will iterate over the entire array (in C-contiguous style with the last index varying the fastest). This iterator object can also be indexed using basic slicing or advanced indexing as long as the selection object is not a tuple. This should be clear from the fact that `x.flat` is a 1-dimensional view. It can be used for integer indexing with 1-dimensional C-style-flat indices. The shape of any returned array is therefore the shape of the integer indexing object.

1.5 Standard array subclasses

The `ndarray` in NumPy is a “new-style” Python built-in-type. Therefore, it can be inherited from (in Python or in C) if desired. Therefore, it can form a foundation for many useful classes. Often whether to sub-class the array object or to simply use the core array component as an internal part of a new class is a difficult decision, and can be simply a matter of choice. NumPy has several tools for simplifying how your new object interacts with other array objects, and so the choice may not be significant in the end. One way to simplify the question is by asking yourself if the object you are interested in can be replaced as a single array or does it really require two or more arrays at its core.

Note that `asarray` always returns the base-class `ndarray`. If you are confident that your use of the array object can handle any subclass of an `ndarray`, then `asanyarray` can be used to allow subclasses to propagate more cleanly through your subroutine. In principal a subclass could redefine any aspect of the array and therefore, under strict guidelines, `asanyarray` would rarely be useful. However, most subclasses of the array object will not redefine certain aspects of the array object such as the buffer interface, or the attributes of the array. One important example, however, of why your subroutine may not be able to handle an arbitrary subclass of an array is that matrices redefine the `*=` operator to be matrix-multiplication, rather than element-by-element multiplication.

1.5.1 Special attributes and methods

See Also:

* Subclassing ndarray (in NumPy User Guide)

NumPy provides several hooks that subclasses of `ndarray` can customize:

```python
__array_finalize__(self)
```

This method is called whenever the system internally allocates a new array from `obj`, where `obj` is a subclass (subtype) of the `ndarray`. It can be used to change attributes of `self` after construction (so as to ensure a 2-d matrix for example), or to update meta-information from the “parent.” Subclasses inherit a default implementation of this method that does nothing.
__array_prepare__(array, context=None)
At the beginning of every ufunc, this method is called on the input object with the highest array priority, or the output object if one was specified. The output array is passed in and whatever is returned is passed to the ufunc. Subclasses inherit a default implementation of this method which simply returns the output array unmodified. Subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the ufunc for computation.

__array_wrap__(array, context=None)
At the end of every ufunc, this method is called on the input object with the highest array priority, or the output object if one was specified. The ufunc-computed array is passed in and whatever is returned is passed to the user. Subclasses inherit a default implementation of this method, which transforms the array into a new instance of the object’s class. Subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the user.

__array_priority__
The value of this attribute is used to determine what type of object to return in situations where there is more than one possibility for the Python type of the returned object. Subclasses inherit a default value of 1.0 for this attribute.

__array__(dtype)
If a class having the __array__ method is used as the output object of an ufunc, results will be written to the object returned by __array__.

1.5.2 Matrix objects

matrix objects inherit from the ndarray and therefore, they have the same attributes and methods of ndarrays. There are six important differences of matrix objects, however, that may lead to unexpected results when you use matrices but expect them to act like arrays:

1. Matrix objects can be created using a string notation to allow Matlab- style syntax where spaces separate columns and semicolons (‘;’) separate rows.

2. Matrix objects are always two-dimensional. This has far-reaching implications, in that m.ravel() is still two-dimensional (with a 1 in the first dimension) and item selection returns two-dimensional objects so that sequence behavior is fundamentally different than arrays.

3. Matrix objects over-ride multiplication to be matrix-multiplication. Make sure you understand this for functions that you may want to receive matrices. Especially in light of the fact that asanyarray(m) returns a matrix when m is a matrix.

4. Matrix objects over-ride power to be matrix raised to a power. The same warning about using power inside a function that uses asanyarray(...) to get an array object holds for this fact.

5. The default __array_priority__ of matrix objects is 10.0, and therefore mixed operations with ndarrays always produce matrices.

6. Matrices have special attributes which make calculations easier. These are

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix.T</td>
<td>transpose</td>
</tr>
<tr>
<td>matrix.H</td>
<td>hermitian (conjugate) transpose</td>
</tr>
<tr>
<td>matrix.I</td>
<td>inverse</td>
</tr>
<tr>
<td>matrix.A</td>
<td>base array</td>
</tr>
</tbody>
</table>

Warning: Matrix objects over-ride multiplication, ‘*’, and power, ‘**’, to be matrix-multiplication and matrix power, respectively. If your subroutine can accept sub-classes and you do not convert to base-class arrays, then you must use the ufuncs multiply and power to be sure that you are performing the correct operation for all inputs.
The matrix class is a Python subclass of the ndarray and can be used as a reference for how to construct your own subclass of the ndarray. Matrices can be created from other matrices, strings, and anything else that can be converted to an ndarray. The name “mat” is an alias for “matrix” in NumPy.

```python
class matrix():
    Returns a matrix from an array-like object, or from a string of data. A matrix is a specialized 2-d array that retains its 2-d nature through operations. It has certain special operators, such as * (matrix multiplication) and ** (matrix power).

Parameters
    data : array_like or string
        If data is a string, the string is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.
    dtype : data-type
        Data-type of the output matrix.
    copy : bool
        If data is already an ndarray, then this flag determines whether the data is copied, or whether a view is constructed.

See Also:
    array

Examples

>>> a = np.matrix('1 2; 3 4')
>>> print a
[[1 2]
 [3 4]]

>>> np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])

Methods

asmatrix(data, dtype=None)
Interpret the input as a matrix.

Unlike matrix, asmatrix does not make a copy if the input is already a matrix or an ndarray. Equivalent to matrix(data, copy=False).

Parameters
    data : array_like
        Input data.

Returns
    mat : matrix
```

1.5. Standard array subclasses
data interpreted as a matrix.

Examples

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

>>> m = np.asmatrix(x)

>>> x[0,0] = 5

>>> m
matrix([[5, 2],
        [3, 4]])
```

*bmat*(obj, ldict=None, gdict=None)

Build a matrix object from a string, nested sequence, or array.

**Parameters**

obj : string, sequence or array

Input data. Variables names in the current scope may be referenced, even if *obj* is a string.

**Returns**

out : matrix

Returns a matrix object, which is a specialized 2-D array.

See Also:

matrix

Examples

```python
>>> A = np.mat('1 1; 1 1')
>>> B = np.mat('2 2; 2 2')
>>> C = np.mat('3 4; 5 6')
>>> D = np.mat('7 8; 9 0')

All the following expressions construct the same block matrix:

```python
>>> np.bmat([[A, B], [C, D]])
```

Example 1: Matrix creation from a string
>>> a = mat([1 2 3; 4 5 3])
>>> print (a*a.T).I
[[ 0.2924 -0.1345]
 [-0.1345  0.0819]]

Example 2: Matrix creation from nested sequence

>>> mat([[1,5,10], [1.0,3,4j]])
matrix([[ 1.+0.j, 5.+0.j, 10.+0.j],
     [ 1.+0.j, 3.+0.j, 0.+4.j]])

Example 3: Matrix creation from an array

>>> mat(random.rand(3,3)).T
matrix([[ 0.7699, 0.7922, 0.3294],
     [ 0.2792, 0.0101, 0.9219],
     [ 0.3398, 0.7571, 0.8197]])

1.5.3 Memory-mapped file arrays

Memory-mapped files are useful for reading and/or modifying small segments of a large file with regular layout, without reading the entire file into memory. A simple subclass of the ndarray uses a memory-mapped file for the data buffer of the array. For small files, the over-head of reading the entire file into memory is typically not significant, however for large files using memory mapping can save considerable resources.

Memory-mapped-file arrays have one additional method (besides those they inherit from the ndarray): .flush() which must be called manually by the user to ensure that any changes to the array actually get written to disk.

Note: Memory-mapped arrays use the the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

```
class memmap :
    Create a memory-map to an array stored in a binary file on disk.

    Parameters
    filename : str or file-like object
        The file name or file object to be used as the array data buffer.

    dtype : data-type, optional
        The data-type used to interpret the file contents. Default is uint8.

    mode : {'r+', 'r', 'w+', 'c'}, optional
        The file is opened in this mode:
        'r' Open existing file for reading only.
        'r+' Open existing file for reading and writing.
        'w+' Create or overwrite existing file for reading and writing.
        'c' Copy-on-write: assignments affect data in memory, but changes are not saved to disk. The file on disk is read-only.
```

1.5. Standard array subclasses 65
Default is ‘r+’.

offset : int, optional

In the file, array data starts at this offset. Since offset is measured in bytes, it should be a multiple of the byte-size of dtype. Requires shape=None. The default is 0.

shape : tuple, optional

The desired shape of the array. By default, the returned array will be 1-D with the number of elements determined by file size and data-type.

order : {‘C’, ‘F’}, optional

Specify the order of the ndarray memory layout: C (row-major) or Fortran (column-major). This only has an effect if the shape is greater than 1-D. The default order is ‘C’.

Notes

The memmap object can be used anywhere an ndarray is accepted. Given a memmap `fp`, `isinstance(fp, numpy.ndarray)` returns `True`.

Memory-mapped arrays use the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

Examples

```python
>>> data = np.arange(12, dtype='float32')
>>> data.resize((3,4))

This example uses a temporary file so that doctest doesn’t write files to your directory. You would use a ‘normal’ filename.

>>> from tempfile import mkdtemp
>>> import os.path as path

>>> filename = path.join(mkdtemp(), 'newfile.dat')

Create a memmap with dtype and shape that matches our data:

```python
>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
```  
```python
memmap([[ 0., 0., 0., 0.],
        [ 0., 0., 0., 0.],
        [ 0., 0., 0., 0.]], dtype=float32)
```  
Write data to memmap array:

```python
>>> fp[:] = data[:]
```  
```python
memmap([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]], dtype=float32)
```  
Deletion flushes memory changes to disk before removing the object:

```python
>>> del fp
```  
Load the memmap and verify data was stored:
>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3, 4))
>>> newfp
memmap([[ 0., 1., 2., 3.],
         [ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]], dtype=float32)

Read-only memmap:

>>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3, 4))
>>> fpr.flags.writeable
False

Copy-on-write memmap:

>>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3, 4))
>>> fpc.flags.writeable
True

It's possible to assign to copy-on-write array, but values are only written into the memory copy of the array, and not written to disk:

>>> fpc
memmap([[ 0., 1., 2., 3.],
         [ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]], dtype=float32)
>>> fpc[0,:] = 0
>>> fpc
memmap([[ 0., 0., 0., 0.],
         [ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]], dtype=float32)

File on disk is unchanged:

>>> fpr
memmap([[ 0., 1., 2., 3.],
         [ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]], dtype=float32)

Offset into a memmap:

>>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
>>> fpo
memmap([ 4., 5., 6., 7., 8., 9., 10., 11.], dtype=float32)

Methods

flush()
Write any changes in the array to the file on disk.
For further information, see memmap.

Parameters
None:

1.5. Standard array subclasses
See Also:

memmap

Example:

```python
>>> a = memmap('newfile.dat', dtype=float, mode='w+', shape=1000)
>>> a[10] = 10.0
>>> a[30] = 30.0
>>> del a
>>> b = fromfile('newfile.dat', dtype=float)
>>> print b[10], b[30]
10.0 30.0
>>> a = memmap('newfile.dat', dtype=float)
>>> print a[10], a[30]
10.0 30.0
```

1.5.4 Character arrays (numpy.char)

See Also:

Creating character arrays (numpy.char)

**Note:** The chararray module exists for backwards compatibility with Numarray, it is not recommended for new development. If one needs arrays of strings, use arrays of `dtype object`, `str` or `unicode`.

These are enhanced arrays of either `string` type or `unicode` type. These arrays inherit from the `ndarray`, but specially-define the operations `+`, `*`, and `%` on a (broadcasting) element-by-element basis. These operations are not available on the standard `ndarray` of character type. In addition, the `chararray` has all of the standard `string` (and `unicode`) methods, executing them on an element-by-element basis. Perhaps the easiest way to create a chararray is to use `self.view(chararray)` where `self` is an ndarray of `str` or `unicode` data-type. However, a chararray can also be created using the `numpy.chararray` constructor, or via the `numpy.char.array` function:

```
chararray (shape[, itemsize, unicode, ...])
```

Provides a convenient view on arrays of string and unicode values.

```
core.defchararray.array (obj[, itemsize, copy, ...])
```

Create a `chararray`.

**class chararray ()**

Provides a convenient view on arrays of string and unicode values.

**Note:** The `chararray` class exists for backwards compatibility with Numarray, it is not recommended for new development. If one needs arrays of strings, use arrays of `dtype object`, `str` or `unicode`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

Versus a regular Numpy array of type `str` or `unicode`, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. `endswith`) and infix operators (e.g. `"+", "*", 
"%")

chararrays should be created using `numpy.char.array` or `numpy.char.asarray`, rather than this constructor directly.

This constructor creates the array, using `buffer` (with `offset` and `strides`) if it is not `None`. If `buffer` is `None`, then constructs a new array with `strides` in “C order”, unless both `len(shape) >= 2` and `order='Fortran'`, in which case `strides` is in “Fortran order”.

68 Chapter 1. Array objects
**Parameters**

- **shape**: tuple
  
  Shape of the array.

- **itemsize**: int, optional
  
  Length of each array element, in number of characters. Default is 1.

- **unicode**: bool, optional
  
  Are the array elements of type unicode (True) or string (False). Default is False.

- **buffer**: int, optional
  
  Memory address of the start of the array data. Default is None, in which case a new array is created.

- **offset**: int, optional
  
  Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be >=0.

- **strides**: array_like of ints, optional
  
  Strides for the array (see `ndarray.strides` for full description). Default is None.

- **order**: {'C', 'F'}, optional
  
  The order in which the array data is stored in memory: ‘C’ -> “row major” order (the default), ‘F’ -> “column major” (Fortran) order.

**Examples**

```python
>>> charar = np.chararray((3, 3))
>>> charar[:, :] = 'abc'
>>> charar
chararray([[b'a', b'a', b'a'],
           [b'a', b'a', b'a'],
           [b'a', b'a', b'a']],
dtype='|S1')
```

```python
>>> charar = np.chararray(charar.shape, itemsize=5)
>>> charar[:, :] = 'abc'
>>> charar
chararray([[b'abc', b'abc', b'abc'],
           [b'abc', b'abc', b'abc'],
           [b'abc', b'abc', b'abc']],
dtype='|S5')
```

**Methods**

- `array(obj, itemsize=None, copy=True, unicode=None, order=None)`
  
  Create a `chararray`.

**Note**: This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type `string_` or `unicode_` and use the free functions in `numpy.char` for fast vectorized string operations instead.

Versus a regular Numpy array of type `str` or `unicode`, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. Comparison operators automatically remove whitespace from the end when comparing values.
3. Vectorized string operations are provided as methods (e.g., `str.endswith`) and infix operators (e.g., `+`, `*`, `%`)

**Parameters**

- `obj`: array of str or unicode-like
- `itemsize`: int, optional
  
  `itemsize` is the number of characters per scalar in the resulting array. If `itemsize` is `None`, and `obj` is an object array or a Python list, the `itemsize` will be automatically determined. If `itemsize` is provided and `obj` is of type str or unicode, then the `obj` string will be chunked into `itemsize` pieces.

- `copy`: bool, optional
  
  If true (default), then the object is copied. Otherwise, a copy will only be made if `__array__` returns a copy, if `obj` is a nested sequence, or if a copy is needed to satisfy any of the other requirements (`itemsize`, `unicode`, `order`, etc.).

- `unicode`: bool, optional
  
  When true, the resulting chararray can contain Unicode characters, when false only 8-bit characters. If `unicode` is `None` and `obj` is one of the following:
  
  - a chararray,
  - an ndarray of type `str` or `unicode`
  - a Python str or unicode object,
  
  then the unicode setting of the output array will be automatically determined.

- `order`: {'C', 'F', 'A'}, optional
  
  Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

Another difference with the standard ndarray of str data-type is that the chararray inherits the feature introduced by Numarray that white-space at the end of any element in the array will be ignored on item retrieval and comparison operations.

### 1.5.5 Record arrays (numpy.rec)

See Also:

- Creating record arrays (numpy.rec), Data type routines, Data type objects (dtype).

Numpy provides the `recarray` class which allows accessing the fields of a record/structured array as attributes, and a corresponding scalar data type object `record`.

**class recarray()**

Construct an ndarray that allows field access using attributes.

Arrays may have a data-types containing fields, analagous to columns in a spread sheet. An example is `[ (x, int), (y, float)]`, where each entry in the array is a pair of `(int, float)`. Normally, these attributes
are accessed using dictionary lookups such as `arr['x']` and `arr['y']`. Record arrays allow the fields to be accessed as members of the array, using `arr.x` and `arr.y`.

**Parameters**

- **shape**: tuple
  Shape of output array.

- **dtype**: data-type, optional
  The desired data-type. By default, the data-type is determined from `formats`, `names`, `titles`, `aligned` and `byteorder`.

- **formats**: list of data-types, optional
  A list containing the data-types for the different columns, e.g. `['i4', 'f8', 'i4']`. The `formats` does not support the new convention of using types directly, i.e. `(int, float, int)`. Note that `formats` must be a list, not a tuple. Given that `formats` is somewhat limited, we recommend specifying `dtype` instead.

- **names**: tuple of str, optional
  The name of each column, e.g. `('x', 'y', 'z')`.

- **buf**: buffer, optional
  By default, a new array is created of the given shape and data-type. If `buf` is specified and is an object exposing the buffer interface, the array will use the memory from the existing buffer. In this case, the `offset` and `strides` keywords are available.

**Returns**

- **rec**: recarray
  Empty array of the given shape and type.

**See Also:**

- `rec.fromrecords`
  Construct a record array from data.

- `record`
  Fundamental data-type for `recarray`.

- `format_parser`
  Determine a data-type from formats, names, titles.

**Notes**

This constructor can be compared to `empty`: it creates a new record array but does not fill it with data. To create a record array from data, use one of the following methods:

1. Create a standard ndarray and convert it to a record array, using `arr.view(np.recarray)`
2. Use the `buf` keyword.
3. Use `np.rec.fromrecords`.

**Examples**

Create an array with two fields, `x` and `y`:
```python
>>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', float), ('y', int)])
>>> x
array([(1.0, 2), (3.0, 4)],
      dtype=[('x', '<f8'), ('y', '<i4')])

>>> x['x']
array([ 1., 3.])

View the array as a record array:

>>> x = x.view(np.recarray)

>>> x.x
array([ 1., 3.])

>>> x.y
array([2, 4])

Create a new, empty record array:

>>> np.recarray((2,),
...     dtype=[('x', int), ('y', float), ('z', int)]) #doctest: +SKIP
rec.array([(-1073741821, 1.2249118382103472e-301, 24547520),
            (3471280, 1.2134086255804012e-316, 0)],
          dtype=[('x', '<i4'), ('y', '<f8'), ('z', '<i4')])
```

Methods

class record()

A data-type scalar that allows field access as attribute lookup.

Methods

1.5.6 Masked arrays (numpy.ma)

See Also:
Masked arrays

1.5.7 Standard container class

For backward compatibility and as a standard “container “class, the UserArray from Numeric has been brought over to NumPy and named `numpy.lib.user_array.container`. The container class is a Python class whose self.array attribute is an ndarray. Multiple inheritance is probably easier with `numpy.lib.user_array.container` than with the ndarray itself and so it is included by default. It is not documented here beyond mentioning its existence because you are encouraged to use the ndarray class directly if you can.

```python
numpy.lib.user_array.container
```

class container(data, dtype=None, copy=True)
1.5.8 Array Iterators

Iterators are a powerful concept for array processing. Essentially, iterators implement a generalized for-loop. If \texttt{myiter} is an iterator object, then the Python code:

\begin{verbatim}
for val in myiter:
    ...
    some code involving val
    ...
\end{verbatim}

calls \texttt{val = myiter.next()} repeatedly until \texttt{StopIteration} is raised by the iterator. There are several ways to iterate over an array that may be useful: default iteration, flat iteration, and \( N \)-dimensional enumeration.

**Default iteration**

The default iterator of an ndarray object is the default Python iterator of a sequence type. Thus, when the array object itself is used as an iterator. The default behavior is equivalent to:

\begin{verbatim}
for i in arr.shape[0]:
    val = arr[i]
\end{verbatim}

This default iterator selects a sub-array of dimension \( N - 1 \) from the array. This can be a useful construct for defining recursive algorithms. To loop over the entire array requires \( N \) for-loops.

\begin{verbatim}
>>> a = arange(24).reshape(3,2,4)+10
>>> for val in a:
...    print 'item:', val
item: [[10 11 12 13]
[14 15 16 17]]
item: [[18 19 20 21]
[22 23 24 25]]
item: [[26 27 28 29]
[30 31 32 33]]
\end{verbatim}

**Flat iteration**

\texttt{ndarray.flat}  
A 1-D iterator over the array.

\begin{verbatim}
flat
A 1-D iterator over the array.
\end{verbatim}

This is a \texttt{numpy.flatiter} instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

\texttt{flatten}

Return a copy of the array collapsed into one dimension.

\begin{verbatim}
flatiter
Examples
\end{verbatim}
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])

>>> x.flat[3]
4

>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])

>>> x.T.flat[3]
5

>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])

>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])

As mentioned previously, the flat attribute of ndarray objects returns an iterator that will cycle over the entire array in C-style contiguous order:

```python
>>> for i, val in enumerate(a.flat):
...    if i%5 == 0: print i, val
0 10
5 15
10 20
15 25
20 30
```

Here, I’ve used the built-in enumerate iterator to return the iterator index as well as the value.

### N-dimensional enumeration

The `ndenumerate` function is a multidimensional index iterator.

```python
class ndenumerate(arr)
```

Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

**Parameters**

- `a`: ndarray

  Input array.

**See Also:**

- `ndindex`, `flatiter`
Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> for index, x in np.ndenumerate(a):
...    print index, x
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

Methods

Sometimes it may be useful to get the N-dimensional index while iterating. The ndenumerate iterator can achieve this.

```python
>>> for i, val in ndenumerate(a):
...    if sum(i)%5 == 0: print i, val
(0, 0, 0) 10
(1, 1, 3) 29
(2, 0, 3) 29
(2, 1, 2) 32
```

Iterator for broadcasting

```python
class broadcast
Produce an object that mimics broadcasting.

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

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

Examples

Manually adding two vectors, using broadcasting:

```python
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]
>>> out
array([[ 5.,  6.,  7.],
       [ 6.,  7.,  8.],
       [ 7.,  8.,  9.]])
```
```

Compare against built-in broadcasting:
Methods

The general concept of broadcasting is also available from Python using the `broadcast` iterator. This object takes N objects as inputs and returns an iterator that returns tuples providing each of the input sequence elements in the broadcasted result.

```python
>>> for val in broadcast([[1,0],[2,3]],[0,1]):
...     print val
    (1, 0)
    (0, 1)
    (2, 0)
    (3, 1)
```

1.6 Masked arrays

Masked arrays are arrays that may have missing or invalid entries. The `numpy.ma` module provides a nearly work-alike replacement for `numpy` that supports data arrays with masks.

1.6.1 The `numpy.ma` module

Rationale

Masked arrays are arrays that may have missing or invalid entries. The `numpy.ma` module provides a nearly work-alike replacement for `numpy` that supports data arrays with masks.

What is a masked array?

In many circumstances, datasets can be incomplete or tainted by the presence of invalid data. For example, a sensor may have failed to record a data, or recorded an invalid value. The `numpy.ma` module provides a convenient way to address this issue, by introducing masked arrays.

A masked array is the combination of a standard `numpy.ndarray` and a mask. A mask is either `nomask`, indicating that no value of the associated array is invalid, or an array of booleans that determines for each element of the associated array whether the value is valid or not. When an element of the mask is `False`, the corresponding element of the associated array is valid and is said to be unmasked. When an element of the mask is `True`, the corresponding element of the associated array is said to be masked (invalid).

The package ensures that masked entries are not used in computations.

As an illustration, let’s consider the following dataset:

```python
>>> import numpy as np
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
```
We wish to mark the fourth entry as invalid. The easiest is to create a masked array:

```python
>>> mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
```

We can now compute the mean of the dataset, without taking the invalid data into account:

```python
>>> mx.mean()
2.75
```

**The `numpy.ma` module**

The main feature of the `numpy.ma` module is the `MaskedArray` class, which is a subclass of `numpy.ndarray`. The class, its attributes and methods are described in more details in the `MaskedArray class` section.

The `numpy.ma` module can be used as an addition to `numpy`:

```python
>>> import numpy as np
>>> import numpy.ma as ma
```

To create an array with the second element invalid, we would do:

```python
>>> y = ma.array([1, 2, 3], mask = [0, 1, 0])
```

To create a masked array where all values close to 1.e20 are invalid, we would do:

```python
>>> z = masked_values([1.0, 1.e20, 3.0, 4.0], 1.e20)
```

For a complete discussion of creation methods for masked arrays please see section *Constructing masked arrays*.

### 1.6.2 Using `numpy.ma`

**Constructing masked arrays**

There are several ways to construct a masked array.

- A first possibility is to directly invoke the `MaskedArray` class.
- A second possibility is to use the two masked array constructors, `array` and `masked_array`.

```python
array (data[, dtype, copy, order, ...])
masked_array

array (data, dtype=None, copy=False, order=False, mask=False, fill_value=None, keep_mask=True, hard_mask=False, shrink=True, subok=True, ndmin=0)
```

An array class with possibly masked values.

An array class with possibly masked values.

An array class with possibly masked values.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data</code></td>
<td>array_like</td>
</tr>
</tbody>
</table>

Masked values of True exclude the corresponding element from any computation.

Construction:

```python
x = MaskedArray(data, mask=nomask, dtype=None, copy=True, fill_value=None, keep_mask=True, hard_mask=False, shrink=True)
```
Input data.

**mask** : sequence, optional

Mask. Must be convertible to an array of booleans with the same shape as `data`. True indicates a masked (i.e. invalid) data.

**dtype** : dtype, optional

Data type of the output. If `dtype` is None, the type of the data argument (`data.dtype`) is used. If `dtype` is not None and different from `data.dtype`, a copy is performed.

**copy** : bool, optional

Whether to copy the input data (True), or to use a reference instead. Default is False.

**subok** : bool, optional

Whether to return a subclass of `MaskedArray` if possible (True) or a plain `MaskedArray`. Default is True.

**ndmin** : int, optional

Minimum number of dimensions. Default is 0.

**fill_value** : scalar, optional

Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

**keep_mask** : bool, optional

Whether to combine `mask` with the mask of the input data, if any (True), or to use only `mask` for the output (False). Default is True.

**hard_mask** : bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be unmasked. Default is False.

**shrink** : bool, optional

Whether to force compression of an empty mask. Default is True.

**masked_array**

alias of `MaskedArray`

A third option is to take the view of an existing array. In that case, the mask of the view is set to `nomask` if the array has no named fields, or an array of boolean with the same structure as the array otherwise.

```python
>>> x = np.array([1, 2, 3])
>>> x.view(ma.MaskedArray)
masked_array(data = [1 2 3],
             mask = False,
             fill_value = 999999)
>>> x = np.array([(1, 1.), (2, 2.)], dtype=[('a',int), ('b', float)])
>>> x.view(ma.MaskedArray)
masked_array(data = [(1, 1.0) (2, 2.0)],
             mask = [(False, False) (False, False)],
             fill_value = (999999, 1e+20),
             dtype = [(‘a’, ‘<i4’), (‘b’, ‘<f8’)])
```
Yet another possibility is to use any of the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asarray(a[, dtype, order])</code></td>
<td>Convert the input to a masked array of the given data-type. If $a$ is a subclass of <code>MaskedArray</code>, a base class <code>MaskedArray</code> is returned.</td>
</tr>
<tr>
<td><code>asanyarray(a[, dtype])</code></td>
<td>Similar to <code>asarray</code>, but conserves subclasses.</td>
</tr>
<tr>
<td><code>fix_invalid(a[, mask, copy, fill_value])</code></td>
<td>Convert the input to a masked array, conserving subclasses. Return input with invalid data masked and replaced by a fill value.</td>
</tr>
<tr>
<td><code>masked_equal(x, value[, copy])</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>masked_greater(x, value[, copy])</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
<tr>
<td><code>masked_greater_equal(x, value[, copy])</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>masked_inside(x, v1, v2[, copy])</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>masked_invalid(a[, copy])</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>masked_less(x, value[, copy])</code></td>
<td>Mask an array where less than a given value.</td>
</tr>
<tr>
<td><code>masked_less_equal(x, value[, copy])</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>masked_not_equal(x, value[, copy])</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>masked_object(x, value[, copy, shrink])</code></td>
<td>Mask the array $x$ where the data are exactly equal to value.</td>
</tr>
<tr>
<td><code>masked_outside(x, v1, v2[, copy])</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>masked_values(x, value[, rtol, atol, copy, ...])</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>masked_where(condition, a[, copy])</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
</tbody>
</table>

**Examples**

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asarray(x)
MaskedArray(data =
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]],
mask = False, fill_value=999999)
```
mask =
False,
fill_value = 1e+20)
>>> type(np.ma.asarray(x))
<class 'numpy.ma.core.MaskedArray'>

asanyarray (a, dtype=None)
Convert the input to a masked array, conserving subclasses.
If a is a subclass of MaskedArray, its class is conserved. No copy is performed if the input is already an
ndarray.

Parameters
a : array_like
Input data, in any form that can be converted to an array.
dtype : dtype, optional
By default, the data-type is inferred from the input data.
order : {'C', 'F'}, optional
Whether to use row-major ('C') or column-major ('FORTRAN') memory representa-
tion. Default is 'C'.

Returns
out : MaskedArray
MaskedArray interpretation of a.

See Also:

asarray
Similar to asanyarray, but does not conserve subclass.

Examples

>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
[ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asanyarray(x)
mixed_array(data =
[[ 0.  1.  2.  3.  4.]
[ 5.  6.  7.  8.  9.]],
mask =
False,
fill_value = 1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>

generate_invalid (a, mask=False, copy=True, fill_value=None)
Return input with invalid data masked and replaced by a fill value.
Invalid data means values of nan, inf, etc.

Parameters
a : array_like
Input array, a (subclass of) ndarray.

Examples
copy : bool, optional

Whether to use a copy of a (True) or to fix a in place (False). Default is True.

fill_value : scalar, optional

Value used for fixing invalid data. Default is None, in which case the a.fill_value
is used.

Returns

b : MaskedArray

The input array with invalid entries fixed.

Notes

A copy is performed by default.

Examples

```python
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [ -- -1.0 nan inf],
             mask = [ True False False False],
             fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data = [ -- -1.0 -- --],
             mask = [ True False True True],
             fill_value = 1e+20)
>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([1.00000000e+00, -1.00000000e+00, 1.00000000e+20, 1.00000000e+20])
>>> x.data
array([1., -1., NaN, Inf])
```

**masked_equal** *(x, value, copy=True)*

Mask an array where equal to a given value.

This function is a shortcut to masked_where, with condition = (x == value). For floating point arrays, consider using masked_values(x, value).

See Also:

- masked_where
- masked_values

Examples

```python
>>> import numpy.ma as ma

>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
             mask = [False False True False],
             fill_value = 1e+20)
```
mask = [False False True False],
     fill_value=999999)

**masked_greater**(x, value, copy=True)

Mask an array where greater than a given value.
This function is a shortcut to **masked_where**, with condition = (x > value).

See Also:

**masked_where**

Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data = [0 1 2 --],
             mask = [False False False True],
             fill_value=999999)
```

**masked_greater_equal**(x, value, copy=True)

Mask an array where greater than or equal to a given value.
This function is a shortcut to **masked_where**, with condition = (x >= value).

See Also:

**masked_where**

Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 -- --],
              mask = [False False True True],
              fill_value=999999)
```

**masked_inside**(x, v1, v2, copy=True)

Mask an array inside a given interval.
Shortcut to **masked_where**, where condition is True for x inside the interval [v1,v2] (v1 <= x <= v2).
The boundaries v1 and v2 can be given in either order.

See Also:

**masked_where**

Mask where a condition is met.
Notes

The array $x$ is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
             fill_value=1e+20)
```

The order of $v1$ and $v2$ doesn’t matter.

```python
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
             fill_value=1e+20)
```

**masked_invalid**($a$, $copy=True$)

Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to `masked_where`, with condition = ~(np.isfinite(a)). Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

**See Also:**

**masked_where**

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0.,  1.,  NaN,  Inf,  4.])
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
             mask = [False False True True False],
             fill_value=1e+20)
```

**masked_less**($x$, $value$, $copy=True$)

Mask an array where less than a given value.

This function is a shortcut to `masked_where`, with condition = (x < value).

**See Also:**

**masked_where**

Mask where a condition is met.
Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
masked_array(data = [-- -- 2 3],
              mask = [ True True False False],
              fill_value=999999)

masked_less_equal(x, value, copy=True)
Mask an array where less than or equal to a given value.
This function is a shortcut to masked_where, with condition = (x <= value).
See Also:

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
masked_array(data = [-- -- -- 3],
              mask = [ True True True False],
              fill_value=999999)

masked_not_equal(x, value, copy=True)
Mask an array where not equal to a given value.
This function is a shortcut to masked_where, with condition = (x != value).
See Also:

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data = [-- -- 2 --],
              mask = [ True True False True],
              fill_value=999999)

masked_object(x, value, copy=True, shrink=True)
Mask the array x where the data are exactly equal to value.
This function is similar to masked_values, but only suitable for object arrays: for floating point, use masked_values instead.
Parameters
x : array_like
Array to mask
value : object
Comparison value
copy : {True, False}, optional
Whether to return a copy of x.
shrink : {True, False}, optional
Whether to collapse a mask full of False to nomask

Returns
result : MaskedArray
The result of masking x where equal to value.

See Also:
masked_where
Mask where a condition is met.
masked_equal
Mask where equal to a given value (integers).
masked_values
Mask using floating point equality.

Examples
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don’t eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> print eat
[-- ham]
>>> # plain ol’ ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> print eat
[cheese ham pineapple]

Note that mask is set to nomask if possible.

>>> eat
masked_array(data = [cheese ham pineapple],
             mask = False,
             fill_value=?)

masked_outside(x, v1, v2, copy=True)
Mask an array outside a given interval.

Shortcut to masked_where, where condition is True for x outside the interval [v1,v2] (x < v1)(x > v2). The boundaries v1 and v2 can be given in either order.

See Also:
**masked_where**

Mask where a condition is met.

**Notes**

The array \( x \) is prefilled with its filling value.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
             mask = [ True True False False True True],
             fill_value=1e+20)
```

The order of \( v1 \) and \( v2 \) doesn’t matter.

```python
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
             mask = [ True True False False True True],
             fill_value=1e+20)
```

**masked_values**

Mask using floating point equality.

Return a MaskedArray, masked where the data in array \( x \) are approximately equal to \( value \), i.e. where the following condition is True

\[
\text{abs}(x - value) \leq \text{atol} + \text{rtol} \times \text{abs}(value)
\]

The fill value is set to \( value \) and the mask is set to nomask if possible. For integers, consider using masked_equal.

**Parameters**

- \( x \) : array_like
  - Array to mask.
- \( value \) : float
  - Masking value.
- \( rtol \) : float, optional
  - Tolerance parameter.
- \( atol \) : float, optional
  - Tolerance parameter (1e-8).
- \( copy \) : bool, optional
  - Whether to return a copy of \( x \).
- \( shrink \) : bool, optional
  - Whether to collapse a mask full of False to nomask.

**Returns**

- \( \text{result} \) : MaskedArray
  - The result of masking \( x \) where approximately equal to \( value \).
See Also:

**masked_where**
Mask where a condition is met.

**masked_equal**
Mask where equal to a given value (integers).

Examples

```python
>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
             mask = [False True False True False],
             fill_value=1.1)
```

Note that `mask` is set to `nomask` if possible.

```python
>>> ma.masked_values(x, 1.5)
masked_array(data = [1. 1.1 2. 1.1 3.],
             mask = False,
             fill_value=1.5)
```

For integers, the fill value will be different in general to the result of `masked_equal`.

```python
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data = [0 1 -- 3 4],
             mask = [False False True False False],
             fill_value=2)
>>> ma.masked_equal(x, 2)
masked_array(data = [0 1 -- 3 4],
             mask = [False False True False False],
             fill_value=999999)
```

**masked_where**(condition, a, copy=True)
Mask an array where a condition is met.

Return `a` as an array masked where `condition` is True. Any masked values of `a` or `condition` are also masked in the output.

Parameters

- **condition**: array_like
  Masking condition. When `condition` tests floating point values for equality, consider using `masked_values` instead.

- **a**: array_like
  Array to mask.

- **copy**: bool
  If True (default) make a copy of `a` in the result. If False modify `a` in place and return a view.

1.6. Masked arrays
Returns

result : MaskedArray

The result of masking a where condition is True.

See Also:

masked_values
    Mask using floating point equality.

masked_equal
    Mask where equal to a given value.

masked_not_equal
    Mask where not equal to a given value.

masked_less_equal
    Mask where less than or equal to a given value.

masked_greater_equal
    Mask where greater than or equal to a given value.

masked_less
    Mask where less than a given value.

masked_greater
    Mask where greater than a given value.

masked_inside
    Mask inside a given interval.

masked_outside
    Mask outside a given interval.

masked_invalid
    Mask invalid values (NaNs or infs).

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
             mask = [ True True True False],
             fill_value=999999)

Mask array b conditional on a.

>>> b = [‘a’, ‘b’, ‘c’, ‘d’]
>>> ma.masked_where(a == 2, b)
mixed_array(data = [a b -- d],
            mask = [False False True False],
            fill_value=N/A)

Effect of the copy argument.
>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data = [-- -- -- 3],
    mask = [ True True True False],
    fill_value=999999)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
    mask = [False True True False],
    fill_value=999999)

>>> a
array([0, 1, 2, 3])

>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c
masked_array(data = [99 -- -- 3],
    mask = [False True True False],
    fill_value=999999)

>>> a
array([99, 1, 2, 3])

When condition or a contain masked values.

>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data = [0 1 -- 3],
    mask = [False False True False],
    fill_value=999999)

>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
>>> b
masked_array(data = [-- 1 2 3],
    mask = [ True False False False],
    fill_value=999999)

>>> ma.masked_where(a == 3, b)
masked_array(data = [-- 1 -- --],
    mask = [ True False True True],
    fill_value=999999)

Accessing the data

The underlying data of a masked array can be accessed in several ways:

- through the data attribute. The output is a view of the array as a numpy.ndarray or one of its subclasses, depending on the type of the underlying data at the masked array creation.
- through the __array__ method. The output is then a numpy.ndarray.
- by directly taking a view of the masked array as a numpy.ndarray or one of its subclass (which is actually what using the data attribute does).
- by using the getdata function.

None of these methods is completely satisfactory if some entries have been marked as invalid. As a general rule, where a representation of the array is required without any masked entries, it is recommended to fill the array with the filled method.
Accessing the mask

The mask of a masked array is accessible through its `mask` attribute. We must keep in mind that a `True` entry in the mask indicates an `invalid` data.

Another possibility is to use the `getmask` and `getmaskarray` functions. `getmask(x)` outputs the mask of `x` if `x` is a masked array, and the special value `nomask` otherwise. `getmaskarray(x)` outputs the mask of `x` if `x` is a masked array. If `x` has no invalid entry or is not a masked array, the function outputs a boolean array of `False` with as many elements as `x`.

Accessing only the valid entries

To retrieve only the valid entries, we can use the inverse of the mask as an index. The inverse of the mask can be calculated with the `numpy.logical_not` function or simply with the `~` operator:

```python
>>> x = ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
>>> x[~x.mask]
masked_array(data = [1 4],
    mask = [False False],
    fill_value = 999999)
```

Another way to retrieve the valid data is to use the `compressed` method, which returns a one-dimensional `ndarray` (or one of its subclasses, depending on the value of the `baseclass` attribute):

```python
>>> x.compressed()
array([1, 4])
```

Note that the output of `compressed` is always 1D.

Modifying the mask

Masking an entry

The recommended way to mark one or several specific entries of a masked array as invalid is to assign the special value `masked` to them:

```python
>>> x = ma.array([1, 2, 3])
>>> x[0] = ma.masked
>>> x
masked_array(data = [-- 2 3],
    mask = [ True False False],
    fill_value = 999999)
```

```python
>>> y = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> y[(0, 1, 2), (1, 2, 0)] = ma.masked
>>> y
masked_array(data =
    [[1 -- 3]
     [4 5 --]
     [-- 8 9]],
    mask =
    [[False True False]
     [False False True]
     [ True False False]],
    fill_value = 999999)
```
NumPy Reference, Release 1.4.1

>>> z[:-2] = ma.masked
>>> z
masked_array(data = [-- -- 3 4],
             mask = [ True  True False False],
             fill_value = 999999)

A second possibility is to modify the mask directly, but this usage is discouraged.

Note: When creating a new masked array with a simple, non-structured datatype, the mask is initially set to the special value nomask, that corresponds roughly to the boolean False. Trying to set an element of nomask will fail with a TypeError exception, as a boolean does not support item assignment.

All the entries of an array can be masked at once by assigning True to the mask:

>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x.mask = True
>>> x
masked_array(data = [-- -- --],
             mask = [ True  True  True],
             fill_value = 999999)

Finally, specific entries can be masked and/or unmasked by assigning to the mask a sequence of booleans:

>>> x = ma.array([1, 2, 3])
>>> x.mask = [0, 1, 0]
>>> x
masked_array(data = [1 -- 3],
             mask = [False  True False],
             fill_value = 999999)

Unmasking an entry

To unmask one or several specific entries, we can just assign one or several new valid values to them:

>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data = [1 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 5],
             mask = [False False False],
             fill_value = 999999)

Note: Unmasking an entry by direct assignment will silently fail if the masked array has a hard mask, as shown by the hardmask attribute. This feature was introduced to prevent overwriting the mask. To force the unmasking of an entry where the array has a hard mask, the mask must first be softened using the soften_mask method before the allocation. It can be re-hardened with harden_mask:

>>> x = ma.array([1, 2, 3], mask=[0, 0, 1], hard_mask=True)
>>> x
masked_array(data = [1 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 5],
             mask = [False False False],
             fill_value = 999999)
masked_array(data = [1 2 --],
    mask = [False False True],
    fill_value = 999999)
>>> x.soften_mask()
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 --],
    mask = [False False True],
    fill_value = 999999)
>>> x.harden_mask()

To unmask all masked entries of a masked array (provided the mask isn’t a hard mask), the simplest solution is to assign the constant `nomask` to the mask:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data = [1 2 --],
    mask = [False False True],
    fill_value = 999999)
>>> x.mask = ma.nomask
>>> x
masked_array(data = [1 2 3],
    mask = [False False False],
    fill_value = 999999)
```

### Indexing and Slicing

As a `MaskedArray` is a subclass of `numpy.ndarray`, it inherits its mechanisms for indexing and slicing.

When accessing a single entry of a masked array with no named fields, the output is either a scalar (if the corresponding entry of the mask is `False`) or the special value `masked` (if the corresponding entry of the mask is `True`):

```python
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x[0]
1
>>> x[-1]
masked_array(data = --,
    mask = True,
    fill_value = 1e+20)
>>> x[-1] is ma.masked
True
```

If the masked array has named fields, accessing a single entry returns a `numpy.void` object if none of the fields are masked, or a 0d masked array with the same dtype as the initial array if at least one of the fields is masked.

```python
>>> y = ma.masked_array([(1,2), (3, 4)],
...    mask=[[0, 0], [0, 1]],
...    dtype=[('a', int), ('b', int)])
>>> y[0]
(1, 2)
>>> y[-1]
masked_array(data = (3, --),
    mask = (False, True),
    fill_value = (999999, 999999),
    dtype = [('a', '<i4'), ('b', '<i4')])
```
When accessing a slice, the output is a masked array whose `data` attribute is a view of the original data, and whose mask is either `nomask` (if there was no invalid entries in the original array) or a copy of the corresponding slice of the original mask. The copy is required to avoid propagation of any modification of the mask to the original.

```python
>>> x = ma.array([1, 2, 3, 4, 5], mask=[0, 1, 0, 0, 1])
>>> mx = x[:3]
>>> mx
masked_array(data = [1 -- 3],
             mask = [False True False],
             fill_value = 999999)
>>> mx[1] = -1
>>> mx
masked_array(data = [1 -1 3],
             mask = [False True False],
             fill_value = 999999)
>>> x.mask
array([False, True, False, False, True], dtype=bool)
>>> x.data
array([ 1, -1, 3, 4, 5])
```

Accessing a field of a masked array with structured datatype returns a `MaskedArray`.

**Operations on masked arrays**

Arithmetic and comparison operations are supported by masked arrays. As much as possible, invalid entries of a masked array are not processed, meaning that the corresponding `data` entries *should* be the same before and after the operation.

**Warning:** We need to stress that this behavior may not be systematic, that masked data may be affected by the operation in some cases and therefore users should not rely on this data remaining unchanged.

The `numpy.ma` module comes with a specific implementation of most ufuncs. Unary and binary functions that have a validity domain (such as `log` or `divide`) return the `masked` constant whenever the input is masked or falls outside the validity domain:

```python
>>> ma.log([-1, 0, 1, 2])
masked_array(data = [-- -- 0.0 0.69314718056],
             mask = [ True True False False],
             fill_value = 1e+20)
```

Masked arrays also support standard numpy ufuncs. The output is then a masked array. The result of a unary ufunc is masked wherever the input is masked. The result of a binary ufunc is masked wherever any of the input is masked. If the ufunc also returns the optional context output (a 3-element tuple containing the name of the ufunc, its arguments and its domain), the context is processed and entries of the output masked array are masked wherever the corresponding input fall outside the validity domain:

```python
>>> x = ma.array([-1, 1, 0, 2, 3], mask=[0, 0, 0, 0, 1])
>>> np.log(x)
masked_array(data = [-- -- 0.0 0.69314718056 --],
             mask = [ True True False False True],
             fill_value = 1e+20)
```
1.6.3 Examples

Data with a given value representing missing data

Let’s consider a list of elements, \( x \), where values of -9999. represent missing data. We wish to compute the average value of the data and the vector of anomalies (deviations from the average):

```python
>>> import numpy.ma as ma

>>> x = [0.,1.,-9999.,3.,4.]

>>> mx = ma.masked_values (x, -9999.)

>>> print mx.mean()
2.0

>>> print mx - mx.mean()
[-2.0 -1.0 -- 1.0 2.0]

>>> print mx.anom()
[-2.0 -1.0 -- 1.0 2.0]
```

Filling in the missing data

Suppose now that we wish to print that same data, but with the missing values replaced by the average value.

```python
>>> print mx.filled(mx.mean())
[ 0. 1. 2. 3. 4.]
```

Numerical operations

Numerical operations can be easily performed without worrying about missing values, dividing by zero, square roots of negative numbers, etc.:

```python
>>> import numpy as np, numpy.ma as ma

>>> x = ma.array([1., -1., 3., 4., 5., 6.], mask=[0,0,0,0,1,0])

>>> y = ma.array([1., 2., 0., 4., 5., 6.], mask=[0,0,0,0,0,1])

>>> print np.sqrt(x/y)
[1.0 -- -- 1.0 -- --]
```

Four values of the output are invalid: the first one comes from taking the square root of a negative number, the second from the division by zero, and the last two where the inputs were masked.

Ignoring extreme values

Let’s consider an array \( d \) of random floats between 0 and 1. We wish to compute the average of the values of \( d \) while ignoring any data outside the range \([0.1, 0.9]\):

```python
>>> print ma.masked_outside(d, 0.1, 0.9).mean()
```

1.6.4 Constants of the `numpy.ma` module

In addition to the `MaskedArray` class, the `numpy.ma` module defines several constants.
masked

The masked constant is a special case of MaskedArray, with a float datatype and a null shape. It is used to test whether a specific entry of a masked array is masked, or to mask one or several entries of a masked array:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 1, 0])
>>> x[1] is ma.masked
True
>>> x[-1] = ma.masked
>>> x
masked_array(data = [1 -- --],
             mask = [False True True],
             fill_value = 999999)
```

nomask

Value indicating that a masked array has no invalid entry. nomask is used internally to speed up computations when the mask is not needed.

masked_print_options

String used in lieu of missing data when a masked array is printed. By default, this string is ‘--’.

### 1.6.5 The MaskedArray class

class MaskedArray()

A subclass of ndarray designed to manipulate numerical arrays with missing data.

An instance of MaskedArray can be thought as the combination of several elements:

- The data, as a regular numpy.ndarray of any shape or datatype (the data).
- A boolean mask with the same shape as the data, where a True value indicates that the corresponding element of the data is invalid. The special value nomask is also acceptable for arrays without named fields, and indicates that no data is invalid.
- A fill_value, a value that may be used to replace the invalid entries in order to return a standard numpy.ndarray.

Attributes and properties of masked arrays

See Also:

Array Attributes

data

Returns the underlying data, as a view of the masked array. If the underlying data is a subclass of numpy.ndarray, it is returned as such.

```python
>>> x = ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.data
matrix([[1, 2],
        [3, 4]])
```

The type of the data can be accessed through the baseclass attribute.

mask

Returns the underlying mask, as an array with the same shape and structure as the data, but where all fields are atomically booleans. A value of True indicates an invalid entry.

1.6. Masked arrays
recordmask

Returns the mask of the array if it has no named fields. For structured arrays, returns a ndarray of booleans where entries are True if all the fields are masked, False otherwise:

```python
generate_array = ma.array(([1, 1], [2, 2], [3, 3], [4, 4], [5, 5]),
                         mask=[[0, 0], [1, 0], [1, 1], [0, 1], [0, 0]],
                         dtype=[('a', int), ('b', int)])
generate_mask = generate_array.recordmask
array([False, False, True, False, False], dtype=bool)
```

fill_value

Returns the value used to fill the invalid entries of a masked array. The value is either a scalar (if the masked array has no named fields), or a 0-D ndarray with the same dtype as the masked array if it has named fields.

The default filling value depends on the datatype of the array:

<table>
<thead>
<tr>
<th>datatypes</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>int</td>
<td>999999</td>
</tr>
<tr>
<td>float</td>
<td>1.e20</td>
</tr>
<tr>
<td>complex</td>
<td>1.e20+0j</td>
</tr>
<tr>
<td>object</td>
<td>‘?’</td>
</tr>
<tr>
<td>string</td>
<td>‘N/A’</td>
</tr>
</tbody>
</table>

baseclass

Returns the class of the underlying data.

```python
generate_matrix = ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 0], [1, 0]])
generate_base = generate_matrix.baseclass
<class 'numpy.matrixlib.defmatrix.matrix'>
```

sharedmask

Returns whether the mask of the array is shared between several masked arrays. If this is the case, any modification to the mask of one array will be propagated to the others.

hardmask

Returns whether the mask is hard (True) or soft (False). When the mask is hard, masked entries cannot be unmasked.

As MaskedArray is a subclass of ndarray, a masked array also inherits all the attributes and properties of a ndarray instance.

<table>
<thead>
<tr>
<th>MaskedArray.base</th>
<th>Base object if memory is from some other object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.ctypes</td>
<td>An object to simplify the interaction of the array with the ctypes module.</td>
</tr>
<tr>
<td>MaskedArray.dtype</td>
<td>Data-type of the array’s elements.</td>
</tr>
<tr>
<td>MaskedArray.flags</td>
<td>Information about the memory layout of the array.</td>
</tr>
<tr>
<td>MaskedArray.itemsize</td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td>MaskedArray nbytes</td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td>MaskedArray.ndim</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>MaskedArray.shape</td>
<td>Tuple of array dimensions.</td>
</tr>
<tr>
<td>MaskedArray.size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>MaskedArray.strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</td>
</tr>
<tr>
<td>MaskedArray.imag</td>
<td>Imaginary part.</td>
</tr>
<tr>
<td>MaskedArray.real</td>
<td>Real part</td>
</tr>
<tr>
<td>MaskedArray.flat</td>
<td>Flat version of the array.</td>
</tr>
<tr>
<td>MaskedArray.<strong>array_priority</strong></td>
<td>int(x[1, base]) -&gt; integer</td>
</tr>
</tbody>
</table>
**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):

- **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 writable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

- **shape (c_intp*self.ndim)**: A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype(’p’) on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

- **strides (c_intp*self.ndim)**: A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

- **data_as(obj)**: Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).
• `shape_as(obj)`: Return the shape tuple as an array of some other c-types type. For example:
  ```python
def self.shape_as(c_value_type):
    # Implementation...
```

• `strides_as(obj)`: Return the strides tuple as an array of some other c-types type. For example:
  ```python
def self.strides_as(c_value_type):
    # Implementation...
```

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(cvalue_types.c_value_type)` returns a pointer to memory that is invalid because the array created as `(a+b)` is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, `ct` will hold a reference to the array until `ct` is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

### Examples

```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>
```

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')
```
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 UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling ndarray.setflags.

The array flags cannot be set arbitrarily:
• UPDATEIFCOPY can only be set False.
• ALIGNED can only be set True if the data is truly aligned.
• WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Attributes

itemsize
Length of one array element in bytes.

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
```
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3

`shape`

Tuple of array dimensions.

**Notes**

May be used to “reshape” the array, as long as this would not require a change in the total number of elements

**Examples**

```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)
```

```
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
```

`sizew`

Number of elements in the array.

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

**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)
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
```

```python
>>> 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
```

**imag**

Imaginary part.

**real**

Real part

**flat**

Flat version of the array.

### 1.6.6 MaskedArray methods

**See Also:**

*Array methods*
**Conversion**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__float__()</code></td>
<td>Convert to float.</td>
</tr>
<tr>
<td><code>__hex__()</code></td>
<td>Convert to int.</td>
</tr>
<tr>
<td><code>__int__()</code></td>
<td>Convert to int.</td>
</tr>
<tr>
<td><code>__long__()</code></td>
<td>Convert to int.</td>
</tr>
<tr>
<td><code>__oct__()</code></td>
<td>Convert to int.</td>
</tr>
<tr>
<td><code>view()</code></td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td><code>astype(newtype)</code></td>
<td>Returns a copy of the MaskedArray cast to given newtype.</td>
</tr>
<tr>
<td><code>byteswap(inplace)</code></td>
<td>Swap the bytes of the array elements.</td>
</tr>
<tr>
<td><code>compressed()</code></td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td><code>filled(fill_value)</code></td>
<td>Return a copy of self, where masked values are filled with a fill value.</td>
</tr>
<tr>
<td><code>tofile(fid[, sep, format])</code></td>
<td>Save a masked array to a file in binary format.</td>
</tr>
<tr>
<td><code>tolist([fill_value])</code></td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td><code>torecords()</code></td>
<td>Return the data portion of the masked array as a hierarchical Python list.</td>
</tr>
<tr>
<td><code>tostring([fill_value, order])</code></td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td><code>view(dtype=None, type=None)</code></td>
<td>Return the array data as a string containing the raw bytes in the array.</td>
</tr>
</tbody>
</table>

---

**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 dtype. 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.
Examples

```python
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
Viewing array data using a different type and dtype:

>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print type(y)
<class 'numpy.matrixlib.defmatrix.matrix'>
Creating a view on a structured array so it can be used in calculations

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

>>> xv[0,1] = 20
>>> print x
[(1, 20) (3, 4)]
Using a view to convert an array to a record array:

>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
Views share data:

>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

**astype** (*newtype*)

Returns a copy of the MaskedArray cast to given newtype.

Returns

`output` : MaskedArray

A copy of self cast to input newtype. The returned record shape matches self.shape.

Examples

```python
>>> x = np.ma.array([[1,2,3.1], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1.0 -- 3.1]
 [-- 5.0 --]
 [7.0 -- 9.0]]
>>> print x.astype(int32)
[[1 -- 3]
""
byteswap()

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace: bool, optional :

If True, swap bytes in-place, default is False.

Returns

out: ndarray :

The byteswapped array. If inplace is True, this is a view to self.

Examples

>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

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

compressed()

Return all the non-masked data as a 1-D array.

Returns

data : ndarray

A new ndarray holding the non-masked data is returned.

Notes

The result is not a MaskedArray!

Examples

>>> x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
>>> x.compressed()
array([0, 1])
>>> type(x.compressed())
<type 'numpy.ndarray'>

filled(fill_value=None)

Return a copy of self, where masked values are filled with a fill value.

If fill_value is None, MaskedArray.fill_value is used instead.
Parameters

fill_value : scalar, optional

The value to use for invalid entries. Default is None.

Notes

The result is not a MaskedArray!

Examples

```python
>>> x = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 0, 1], fill_value=-999)
>>> x.filled()
array([1, 2, -999, 4, -999])
>>> type(x.filled())
<type 'numpy.ndarray'>
```

Subclassing is preserved. This means that if the data part of the masked array is a matrix, filled returns a matrix:

```python
>>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.filled()
matrix([[ 1, 999999],
        [999999, 4]])
```

tofile (fid, sep=",", format='%s')

Save a masked array to a file in binary format.

Warning: This function is not implemented yet.

Raises

NotImplementedError :

When tofile is called.

toflex()

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

•the _data field stores the _data part of the array.
•the _mask field stores the _mask part of the array.

Parameters

None :

Returns

record : ndarray

A new flexible-type ndarray with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

Notes

A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value, ...) will be lost.
Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[1 -- 3]
[-- 5 --]
[7 -- 9]

>>> print x.toflex()
[[True, False, False],
 [False, True, False],
 [False, True, False]]

tolist (fill_value=None)

Return the data portion of the masked array as a hierarchical Python list.

Data items are converted to the nearest compatible Python type. Masked values are converted to fill_value. If fill_value is None, the corresponding entries in the output list will be None.

Parameters

- fill_value : scalar, optional
  The value to use for invalid entries. Default is None.

Returns

- result : list
  The Python list representation of the masked array.

Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)

>>> x.tolist()
[[1, None, 3], [None, 5, None], [7, None, 9]]

>>> x.tolist(-999)
[[1, -999, 3], [-999, 5, -999], [7, -999, 9]]

torecords ()

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:
- the _data field stores the _data part of the array.
- the _mask field stores the _mask part of the array.

Parameters

- None :

Returns

- record : ndarray
  A new flexible-type ndarray with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

Notes

A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value, ...) will be lost.
Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
```
```python
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
```
```python
>>> print x.toflex()
[(1, False) (2, True) (3, False)]
[(4, True) (5, False) (6, True)]
[(7, False) (8, True) (9, False)]
```

**tostring** *(fill_value=None, order='C')*

Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

Parameters

- **fill_value**: scalar, optional
  Value used to fill in the masked values. Default is None, in which case `MaskedArray.fill_value` is used.
- **order**: {'C','F','A'}, optional
  Order of the data item in the copy. Default is ‘C’.
  - ‘C’ – C order (row major).
  - ‘F’ – Fortran order (column major).
  - ‘A’ – Any, current order of array.
  - None – Same as ‘A’.

See Also:

- `ndarray.tostring`, `tolist`, `tofile`

Notes

As for `ndarray.tostring`, information about the shape, dtype, etc., but also about `fill_value`, will be lost.

Examples

```python
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tostring()
'\x01\x00\x00\x00?B\x0f\x00?B\x0f\x00\x04\x00\x00\x00'
```

Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with n integers which will be interpreted as an n-tuple.
NumPy Reference, Release 1.4.1

**MaskedArray.flatten**(order)
Return a copy of the array collapsed into one dimension.  

**Parameters**

- **order**: {'C', 'F'}, optional
  Whether to flatten in C (row-major) or Fortran (column-major) order. 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])
```

**ravel()**

Returns a 1D version of self, as a view.

**Returns**

- **MaskedArray**
  Output view is of shape (self.size,) (or (np.ma.product(self.shape),)).

**Examples**

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]]
```
```python
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

**reshape(** *s*, **kwargs**

Give a new shape to the array without changing its data.

Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

**Parameters**

- **shape**: int or tuple of ints
  
  The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

- **order**: {'C', 'F'}, optional
  
  Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

**Returns**

- **reshaped_array**: array
  
  A new view on the array.

**See Also:**

- **reshape**
  
  Equivalent function in the masked array module.

- **numpy.ndarray.reshape**
  
  Equivalent method on ndarray object.

- **numpy.reshape**
  
  Equivalent function in the NumPy module.

**Notes**

The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`

**Examples**

```python
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
>>> print x
[[-- 2]
 [3 --]]
>>> x = x.reshape((4,1))
>>> print x
[[--]
 [2]
 [3]
 [--]]
```

**resize** *(newshape, refcheck=True, order=False)*

**Warning:** This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the `numpy.ma.resize` function instead.
This method is difficult to implement safely and may be deprecated in future releases of NumPy.

**squeeze()**
Remove single-dimensional entries from the shape of *a*.
Refer to `numpy.squeeze` for full documentation.

See Also:
- `numpy.squeeze` equivalent function

**swapaxes()**
Return a view of the array with *axis1* and *axis2* interchanged.
Refer to `numpy.swapaxes` for full documentation.

See Also:
- `numpy.swapaxes` equivalent function

**transpose()**
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]])
```
Item selection and manipulation

For array methods that take an `axis` keyword, it defaults to `None`. If axis is `None`, then the array is treated as a 1-D array. Any other value for `axis` represents the dimension along which the operation should proceed.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argmax</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value <code>fill_value</code>.</td>
</tr>
<tr>
<td><code>argmin</code></td>
<td>Returns array of indices to the minimum values along the given axis. Masked values are treated as if they had the value <code>fill_value</code>.</td>
</tr>
<tr>
<td><code>argsort</code></td>
<td>Returns array of indices that sort the array along the specified axis. Masked values are filled beforehand to <code>fill_value</code>.</td>
</tr>
<tr>
<td><code>choose</code></td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td><code>compress</code></td>
<td>Return <code>a</code> where condition is <code>True</code>.</td>
</tr>
<tr>
<td><code>diagonal</code></td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>Fill the array with a scalar value.</td>
</tr>
<tr>
<td><code>item</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>nonzero</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>put</code></td>
<td>Set storage-indexed locations to corresponding values.</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>searchsorted</code></td>
<td>Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Return a sorted copy of an array.</td>
</tr>
<tr>
<td><code>take</code></td>
<td>Return an array formed from the elements of <code>a</code> at the given indices.</td>
</tr>
</tbody>
</table>

`argmax` *(axis=None, fill_value=None, out=None)*

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value `fill_value`.

**Parameters**

- `axis`: {None, integer}
  
  If None, the index is into the flattened array, otherwise along the specified axis.

- `fill_value`: {var}, optional
  
  Value used to fill in the masked values. If None, the output of `maximum_fill_value(self._data)` is used instead.

- `out`: {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.
Returns

index_array : {integer_array}

Examples

>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])

argmin (axis=None, fill_value=None, out=None)

Return array of indices to the minimum values along the given axis.

Parameters

axis : {None, integer}
If None, the index is into the flattened array, otherwise along the specified axis

fill_value : {var}, optional
Value used to fill in the masked values. If None, the output of minimum_fill_value(self._data) is used instead.

out : {None, array}, optional
Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

{ndarray, scalar} :
If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

Examples

>>> x = np.ma.array(arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> print x
[[-- --]
 [2 3]]
>>> print x.argmin(axis=0, fill_value=-1)
[0 0]
>>> print x.argmin(axis=0, fill_value=9)
[1 1]

argsort (axis=None, fill_value=None, kind='quicksort', order=None)

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

Parameters

axis : int, optional
Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

fill_value : var, optional
Value used to fill the array before sorting. The default is the `fill_value` attribute of the input array.

**Kind** : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm.

**Order** : list, optional

When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**Returns**

index array : ndarray, int

Array of indices that sort `a` along the specified axis. In other words, `a[index_array]` yields a sorted `a`.

See Also:

- **sort**
  Describes sorting algorithms used.

- **lexsort**
  Indirect stable sort with multiple keys.

- **ndarray.sort**
  Inplace sort.

**Notes**

See **sort** for notes on the different sorting algorithms.

**Examples**

```python
>>> a = np.ma.array([3, 2, 1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> a.argsort()
array([1, 0, 2])
```

**choose**

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

**compress** *(condition, axis=None, out=None)*

Return `a` where condition is `True`.

If condition is a `MaskedArray`, missing values are considered as `False`.

**Parameters**

- `condition` : var
Boolean 1-d array selecting which entries to return. If len(condition) is less than the
size of a along the axis, then output is truncated to length of condition array.

axis : {None, int}, optional
    Axis along which the operation must be performed.

out : {None, ndarray}, optional
    Alternative output array in which to place the result. It must have the same shape as the
    expected output but the type will be cast if necessary.

Returns
result : MaskedArray
    A MaskedArray object.

Notes
Please note the difference with compressed! The output of compress has a mask, the output of
compressed does not.

Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
>>> x.compress([1, 0, 1])
mixed_array(data = [1 3],
    mask = [False False],
    fill_value=999999)

>>> x.compress([1, 0, 1], axis=1)
mixed_array(data =
    [[1 3]
     [-- --]
     [7 9]],
    mask =
    [[False False]
     [ True  True]
     [False False]],
    fill_value=999999)
```

diagonal()
Return specified diagonals.

Refer to numpy.diagonal for full documentation.

See Also:

numpy.diagonal
equivalent function

fill()
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.])
```

**item()**

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, \texttt{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 \texttt{item()}, unless fields are defined, in which case a tuple is returned.

**item** is very similar to \texttt{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
```
nonzero()
Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that
dimension. The corresponding non-zero values can be obtained with:

a[a.nonzero()]

To group the indices by element, rather than dimension, use instead:

np.transpose(a.nonzero())

The result of this is always a 2d array, with a row for each non-zero element.

Parameters
None :

Returns

tuple_of_arrays : tuple
Indices of elements that are non-zero.

See Also:

numpy.nonzero
Function operating on ndarrays.

flatnonzero
Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero
Equivalent ndarray method.

Examples

>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
[[ 1.  0.  0.]
[ 0.  1.  0.]
[ 0.  0.  1.]],
mask = False,
fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[ 1.0  0.0  0.0]
[ 0.0 --  0.0]
[ 0.0  0.0  1.0]],
mask =
[[False False False]
[False  True False]
[False  True False]
[False  True False]
Indices can also be grouped by element.

```python
c>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `ma.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```python
c>> a = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
c>> a > 3
masked_array(data =
             [[False False False]
              [ True True True]
              [ True True True]],
            mask =
            False,
            fill_value=999999)
c>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the condition array can also be called.

```python
c>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

### put(indices, values, mode='raise')
Sets storage-indexed locations to corresponding values.

Sets self._data.flat[n] = values[n] for each n in indices. If `values` is shorter than `indices` then it will repeat. If `values` has some masked values, the initial mask is updated in consequence, else the corresponding values are unmasked.

**Parameters**

- **indices**: 1-D array_like
  - Target indices, interpreted as integers.
- **values**: array_like
  - Values to place in self._data copy at target indices.
- **mode**: {'raise', 'wrap', 'clip'}, optional
  - Specifies how out-of-bounds indices will behave. ‘raise’ : raise an error. ‘wrap’ : wrap around. ‘clip’ : clip to the range.

**Notes**

`values` can be a scalar or length 1 array.
Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
>>> x.put([0,4,8],[10,20,30])
>>> print x
[[10 -- 3]
[-- 20 --]
[7 -- 30]]

>>> x.put(4,999)
>>> print x
[[10 -- 3]
[-- 999 --]
[7 -- 30]]
```

repeat()

Repeat elements of an array.

Refer to numpy.repeat for full documentation.

See Also:

numpy.repeat

equivalent function

searchsorted()

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

sort (axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)

Return a sorted copy of an array.

Parameters

- a : array_like
  - Array to be sorted.
- axis : int or None, optional
  - Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.
- kind : {'quicksort', 'mergesort', 'heapsort'}, optional
  - Sorting algorithm. Default is ‘quicksort’.
- order : list, optional
  - When a is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.
endwith : {True, False}, optional

Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

fill_value : {var}

Value used to fill in the masked values. If None, use the output of minimum_fill_value().

Returns

sorted_array : ndarray

Array of the same type and shape as a.

See Also:

ndarray.sort

Method to sort an array in-place.

argsort

Indirect sort.

lexsort

Indirect stable sort on multiple keys.

searchsorted

Find elements in a sorted array.

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>~n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```python
>>> a = np.array([[1, 4], [3, 1]])
>>> np.sort(a)  # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)  # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)  # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the `order` keyword to specify a field to use when sorting a structured array:

```python
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ...
       ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)  # create a structured array
```
```python
>>> np.sort(a, order='height')  # doctest: +SKIP
array([('Galahad', 1.7, 38),  
       ('Arthur', 1.8, 41),  
       ('Lancelot', 1.8999999999999999, 38)],  
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

Sort by age, then height if ages are equal:

```python
>>> np.sort(a, order=['age', 'height'])  # doctest: +SKIP
array([('Galahad', 1.7, 38),  
       ('Lancelot', 1.8999999999999999, 38),  
       ('Arthur', 1.8, 41)],  
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

take()  
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

Pickling and copy

```python
MaskedArray.copy(order)  
Return a copy of the array.

MaskedArray.dump(file)  
Dump a pickle of the array to the specified file. The array can be read back with  
pickle.load or numpy.load.

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

copy()  
Return a copy of the array.

Parameters  
order : {'C', 'F', 'A'}, optional  
By default, the result is stored in C-contiguous (row-major) order in memory. If order is  
F, the result has ‘Fortran’ (column-major) order. If order is ‘A’ (‘Any’), then the result  
has the same order as the input.

Examples

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

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],  
       [0, 0, 0]])
```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags[‘C_CONTIGUOUS’]
True

dump ()
Dumps 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 :

Calculations

MaskedArray.all (self, axis=None, out=None)
Check if all of the elements of a are true.

Perform a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are true.

MaskedArray.anom (self, axis=None, dtype=None)
Computes the (unweighted) anomalies (deviations from the arithmetic mean) along the given axis.

MaskedArray.any (self, axis=None, out=None)
Check if any of the elements of a are true.

MaskedArray.clip (a, a_min, a_max, out=None)
Clip (lim) values to the range [a_min, a_max].

MaskedArray.conj ()
Complex-conjugate all elements.

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

MaskedArray.cumprod (self, axis=None, dtype=None, out=None)
Return the cumulative product of the elements along the given axis. Masked elements are set to 1 internally for computation.

MaskedArray.cumsum (self, axis=None, dtype=None, out=None)
Return the cumulative sum of the elements along the given axis. Masked elements are set to 1 internally for computation.

MaskedArray.max (self, axis=None, out=None, fill_value=None)
Return the maximum along a given axis.

MaskedArray.mean (self, axis=None, dtype=None, out=None)
Returns the average of the array elements.

MaskedArray.min (self, axis=None, out=None, fill_value=None)
Return the minimum along a given axis.

MaskedArray.prod (self, axis=None, dtype=None, out=None)
Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

MaskedArray.product (self, axis=None, dtype=None, out=None)
Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

MaskedArray.ptp (self, axis=None, out=None, fill_value=None)
Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

MaskedArray.round (decimals=0, out=None)
Return an array rounded a to the given number of decimals.

Perform a rounding operation along the specified axis.

MaskedArray.std (self, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
Computes the standard deviation along the specified axis.

Perform a standard deviation operation along the specified axis.

MaskedArray.sum (self, axis=None, dtype=None, out=None, keepdims=False)
Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

Perform a sum operation along the specified axis.

MaskedArray.trace (offset=0, axis1=0, axis2=-1, out=None)
Return the sum along diagonals of the array.

Perform a trace operation along the specified axis.

all (axis=None, out=None)
Check if all of the elements of a are true.

Perform a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are true.

1.6. Masked arrays
are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**

axis : {None, integer}

Axis to perform the operation over. If None, perform over flattened array.

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

all equivalent function

**Examples**

```python
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

**anom (axis=None, dtype=None)**

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**

axis : int, optional

Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional

Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

**See Also:**

mean Compute the mean of the array.

**Examples**

```python
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
              mask = False,
              fill_value = 1e+20)
```

**any (axis=None, out=None)**

Check if any of the elements of a are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.
Parameters

- **axis**: {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array and return a scalar.

- **out**: {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

- **any**
  
  equivalent function

- **clip()**
  
  Return an array whose values are limited to \[a_{\text{min}}, a_{\text{max}}\].

  Refer to `numpy.clip` for full documentation.

  See Also:

  - **numpy.clip**
    
    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

- **cumprod** *(axis=None, dtype=None, out=None)*

  Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

  Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

  Parameters

  - **axis**: {None, -1, int}, optional
    
    Axis along which the product is computed. The default \(axis = \text{None}\) is to compute over the flattened array.

  - **dtype**: {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the `dtype` is the same as that of `a`.

**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

`cumprod` : ndarray

A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

**Notes**

The mask is lost if `out` is not a valid `MaskedArray`!

Arithmetic is modular when using integer types, and no error is raised on overflow.

### cumsum

`cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

`axis` : {None, -1, int}, optional

Axis along which the sum is computed. The default (`axis = None`) is to compute over the flattened array. `axis` may be negative, in which case it counts from the last to the first axis.

`dtype` : {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If `dtype` is not specified, it defaults to the `dtype` of `a`, unless `a` has an integer `dtype` with a precision less than that of the default platform integer. In that case, the default platform integer is used.

`out` : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

`cumsum` : ndarray

A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

**Notes**

The mask is lost if `out` is not a valid `MaskedArray`!

Arithmetic is modular when using integer types, and no error is raised on overflow.
Examples

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

**max** *(axis=None, out=None, fill_value=None)*

Return the maximum along a given axis.

**Parameters**

- **axis**: {None, int}, optional
  Axis along which to operate. By default, `axis` is None and the flattened input is used.

- **out**: array_like, optional
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- **fill_value**: {var}, optional
  Value used to fill in the masked values. If None, use the output of `maximum_fill_value()`.

**Returns**

- **amax**: array_like
  New array holding the result. If `out` was specified, `out` is returned.

**See Also:**

- `maximum_fill_value`
  Returns the maximum filling value for a given datatype.

**mean** *(axis=None, dtype=None, out=None)*

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to `numpy.mean` for the full documentation.

**Parameters**

- **a**: array_like
  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.

- **axis**: int, optional
  Axis along which the means are computed. The default is to compute the mean of the flattened array.

- **dtype**: dtype, optional
  Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- **mean**: ndarray, see dtype parameter above
If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also:

```python
numpy.ma.mean
   Equivalent function.

numpy.mean
   Equivalent function on non-masked arrays.

numpy.ma.average
   Weighted average.
```

Examples

```python
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
             mask = [False False True],
            fill_value = 999999)
>>> a.mean()
1.5
```

```python
min (axis=None, out=None, fill_value=None)
   Return the minimum along a given axis.

   Parameters
   --------
   axis : {None, int}, optional
      Axis along which to operate. By default, axis is None and the flattened input is used.
   out : array_like, optional
      Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
   fill_value : {var}, optional
      Value used to fill in the masked values. If None, use the output of minimum_fill_value.

   Returns
   -------
   amin : array_like
      New array holding the result. If out was specified, out is returned.

   See Also:
   ---------

   minimum_fill_value
      Returns the minimum filling value for a given datatype.

prod (axis=None, dtype=None, out=None)
   Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

   Parameters
   --------
   axis : {None, int}, optional
      Axis over which the product is taken. If None is used, then the product is over all the array elements.
   dtype : {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

```
out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.
```

Returns

```
product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.
```

See Also:

```
prod

equivalent function
```

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> np.prod([1.,2.])
2.0
>>> np.prod([1.,2.], dtype=np.int32)
2
>>> np.prod([[1.,2.],[3.,4.]])
24.0
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])
```

1.6. Masked arrays

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

```
axis : {None, int}, optional

Axis over which the product is taken. If None is used, then the product is over all the array elements.
```

```
dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.
```

```
out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.
```

Returns

```
product_along_axis : {array, scalar}, see dtype parameter above.
```

```
Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

See Also:

**prod**

equivalent function

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> np.prod([1.,2.])
2.0
>>> np.prod([1.,2.], dtype=np.int32)
2
>>> np.prod([[1.,2.],[3.,4.]])
24.0
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])
```

**ptp** *(axis=None, out=None, fill_value=None)*

Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters

- **axis**: {None, int}, optional
  
  Axis along which to find the peaks. If None (default) the flattened array is used.

- **out**: {None, array_like}, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

- **fill_value**: {var}, optional
  
  Value used to fill in the masked values.

Returns

- **ptp**: ndarray.
  
  A new array holding the result, unless out was specified, in which case a reference to out is returned.

**round** *(decimals=0, out=None)*

Return an array rounded a to the given number of decimals.

Refer to numpy.around for full documentation.

See Also:

- **numpy.around**
  
  equivalent function

**std** *(axis=None, dtype=None, out=None, ddof=0)*

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.
Parameters

- **a**: array_like
  Calculate the standard deviation of these values.

- **axis**: int, optional
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- **dtype**: dtype, optional
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

- **ddof**: int, optional
  Means Delta Degrees of Freedom. The divisor used in calculations is \( N - \text{ddof} \), where \( N \) represents the number of elements. By default \( \text{ddof} \) is zero.

Returns

- **standard_deviation**: ndarray, see dtype parameter above.
  If **out** is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

- var, mean
- numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}((x - x.\text{mean()}^2)} \).

The mean is normally calculated as \( x.\text{sum()}/N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of the infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( \text{ddof}=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( \text{std} \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( \text{std} \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1., 1.])
```
>>> np.std(a, axis=1)
array([ 0.5,  0.5])

In single precision, std() can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

**sum** *(axis=None, dtype=None, out=None)*

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

**Parameters**

- **axis**: {None, -1, int}, optional
  Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional
  Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out**: {None, ndarray}, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

- **sum_along_axis**: MaskedArray or scalar
  An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

**Examples**

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[[0] + [1, 0]*4]
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
>>> print x.sum()
25
>>> print x.sum(axis=1)
[4 5 16]
>>> print x.sum(axis=0)
[8 5 12]
>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type 'numpy.int64'>
```
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

var (axis=None, dtype=None, out=None, ddof=0)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- a : array_like
  Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

- axis : int, optional
  Axis along which the variance is computed. The default is to compute the variance of the flattened array.

- dtype : dtype, optional
  Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

- out : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type is cast if necessary.

- ddof : int, optional
  "Delta Degrees of Freedom": the divisor used in calculation is \( N - \text{ddof} \), where \( N \) represents the number of elements. By default \( \text{ddof} = 0 \).

Returns

- variance : ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise a reference to the output array is returned.

See Also:

std
Standard deviation

mean
Average

numpy.doc.ufuncs
Section “Output arguments”
Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x.\text{mean()} ) ** 2) \).

The mean is normally calculated as \( x.\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of the infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a,0)
array([[ 1. , 1. ]])
>>> np.var(a,1)
array([[ 0.25, 0.25]])
```

In single precision, \( \text{var}() \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

Arithmetic and comparison operations

Comparison operators:

<table>
<thead>
<tr>
<th>MaskedArray.<strong>lt</strong>()</th>
<th>x.<strong>lt</strong>(y) ( \iff ) ( x &lt; y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.<strong>le</strong>()</td>
<td>x.<strong>le</strong>(y) ( \iff ) ( x \leq y )</td>
</tr>
<tr>
<td>MaskedArray.<strong>gt</strong>()</td>
<td>x.<strong>gt</strong>(y) ( \iff ) ( x &gt; y )</td>
</tr>
<tr>
<td>MaskedArray.<strong>ge</strong>()</td>
<td>x.<strong>ge</strong>(y) ( \iff ) ( x \geq y )</td>
</tr>
<tr>
<td>MaskedArray.<strong>eq</strong>()(self, other)</td>
<td>Check whether other equals self elementwise</td>
</tr>
<tr>
<td>MaskedArray.<strong>ne</strong>()(self, other)</td>
<td>Check whether other doesn’t equal self elementwise</td>
</tr>
</tbody>
</table>

```python
__lt__()
x.__lt__(y) \( \iff \) \( x < y \)
__le__()
x.__le__(y) \( \iff \) \( x \leq y \)
```
__gt__(y) <==> x>y

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

__eq__(other)
Check whether other equals self elementwise

__ne__(other)
Check whether other doesn’t equal self elementwise

Truth value of an array (bool):

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

__nonzero__()

Add other to self, and return a new masked array.

Add other to self, and return a new masked array.

Subtract other to self, and return a new masked array.

Subtract other to self, and return a new masked array.

Multiply other by self, and return a new masked array.

Multiply other by self, and return a new masked array.

Divide other into self, and return a new masked array.

Divide other into self, and return a new masked array.

Divide other into self, and return a new masked array.

Divide other into self, and return a new masked array.

Raise self to the power other, masking the potential NaNs/Inf's

Add other to self, and return a new masked array.

Add other to self, and return a new masked array.

Subtract other to self, and return a new masked array.

Subtract other to self, and return a new masked array.

Multiply other by self, and return a new masked array.

Multiply other by self, and return a new masked array.

Divide other into self, and return a new masked array.

Divide other into self, and return a new masked array.

Divide other into self, and return a new masked array.

Divide other into self, and return a new masked array.

Raise self to the power other, masking the potential NaNs/Inf's
__radd__(other)
    Add other to self, and return a new masked array.

__sub__(other)
    Subtract other to self, and return a new masked array.

__rsub__(other)
    Subtract other to self, and return a new masked array.

__mul__(other)
    Multiply other by self, and return a new masked array.

__rmul__(other)
    Multiply other by self, and return a new masked array.

__div__(other)
    Divide other into self, and return a new masked array.

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

__truediv__(other)
    Divide other into self, and return a new masked array.

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

__floordiv__(other)
    Divide other into self, and return a new masked array.

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

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

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

__divmod__()

__rdivmod__()

__pow__(other)
    Raise self to the power other, masking the potential NaNs/Infs

__rpow__()

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

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

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

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

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

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

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

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

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

Arithmetic, in-place:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>iadd</strong> (other)</td>
<td>Add other to self in-place.</td>
</tr>
<tr>
<td><strong>isub</strong> (other)</td>
<td>Subtract other from self in-place.</td>
</tr>
<tr>
<td><strong>imul</strong> (other)</td>
<td>Multiply self by other in-place.</td>
</tr>
<tr>
<td><strong>idiv</strong> (other)</td>
<td>Divide self by other in-place.</td>
</tr>
<tr>
<td><strong>itruediv</strong> ()</td>
<td>x.<strong>itruediv</strong>(y) &lt;==&gt; x/y</td>
</tr>
<tr>
<td><strong>ifloordiv</strong> ()</td>
<td>x.<strong>ifloordiv</strong>(y) &lt;==&gt; x//y</td>
</tr>
<tr>
<td><strong>imod</strong> ()</td>
<td>x.<strong>imod</strong>(y) &lt;==&gt; x%y</td>
</tr>
<tr>
<td><strong>ipow</strong> (other)</td>
<td>Raise self to the power other, in place.</td>
</tr>
<tr>
<td><strong>ilshift</strong> ()</td>
<td>x.<strong>ilshift</strong>(y) &lt;==&gt; x&lt;&lt;y</td>
</tr>
<tr>
<td><strong>irshift</strong> ()</td>
<td>x.<strong>irshift</strong>(y) &lt;==&gt; x&gt;&gt;y</td>
</tr>
<tr>
<td><strong>iand</strong> ()</td>
<td>x.<strong>iand</strong>(y) &lt;==&gt; x&amp;y</td>
</tr>
<tr>
<td><strong>ior</strong> ()</td>
<td>x.<strong>ior</strong>(y) &lt;==&gt; x</td>
</tr>
<tr>
<td><strong>ixor</strong> ()</td>
<td>x.<strong>ixor</strong>(y) &lt;==&gt; x^y</td>
</tr>
</tbody>
</table>

1.6. Masked arrays
__irshift__(y) <==> x\gg y

__iand__(y) <==> x\& y

__ior__(y) <==> x\mid y

__ixor__(y) <==> x\oplus y

Representation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.<strong>repr</strong>(self)</td>
<td>Literal string representation.</td>
</tr>
<tr>
<td>MaskedArray.<strong>str</strong>(self)</td>
<td>String representation.</td>
</tr>
<tr>
<td>MaskedArray.ids(self)</td>
<td>Return the addresses of the data and mask areas.</td>
</tr>
<tr>
<td>MaskedArray.iscontiguous(self)</td>
<td>Return a boolean indicating whether the data is contiguous.</td>
</tr>
</tbody>
</table>

__repr__(self)
Literal string representation.

__str__(self)
String representation.

ids(self)
Return the addresses of the data and mask areas.

Parameters
None:

Examples

```python
>>> x = np.ma.array([1, 2, 3], mask=[0, 1, 1])
>>> x.ids()
(166670640, 166659832)
```

If the array has no mask, the address of nomask is returned. This address is typically not close to the data in memory:

```python
>>> x = np.ma.array([1, 2, 3])
>>> x.ids()
(166691080, 3083169284L)
```

iscontiguous(self)
Return a boolean indicating whether the data is contiguous.

Parameters
None:

Examples

```python
>>> x = np.ma.array([1, 2, 3])
>>> x.iscontiguous()
True
```

iscontiguous returns one of the flags of the masked array:
>>> x.flags
   C_CONTIGUOUS : True
   F_CONTIGUOUS : True
   OWNDATA : False
   WRITEABLE : True
   ALIGNED : True
   UPDATEIFCOPY : False

Special methods

For standard library functions:

### MaskedArray.__copy__(order)
Return a copy of the array.

**Parameters**
- **order**: {'C', 'F', 'A'}, optional
  - If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

### MaskedArray.__deepcopy__(self, memo)
Return the internal state of the masked array, for pickling purposes.

### MaskedArray.__getstate__()
Return the internal state of the masked array, for pickling purposes. `state` is typically the output of the `__getstate__` output, and is a 5-tuple:

- class name
- a tuple giving the shape of the data
- a typecode for the data
- a binary string for the data
- a binary string for the mask.

### MaskedArray.__reduce__()
Return a 3-tuple for pickling a MaskedArray.

### MaskedArray.__setstate__(state)
Restore the internal state of the masked array, for pickling purposes. `state` is typically the output of the `__getstate__` output, and is a 5-tuple:

Basic customization:
MaskedArray.__new__(cls[, data, mask, dtype, copy, subok, ndmin, fill_value, keep_mask, hard_mask, flag, shrink, **options)]
Create a new masked array from scratch.
a.__array__(dtype) -> reference if type unchanged, copy otherwise.
Special hook for ufuncs. Wraps the numpy array and sets the mask according to context.

__array__(self, obj[, context])
Special hook for ufuncs. Wraps the numpy array and sets the mask according to context.

Container customization: (see Indexing)

Specific methods
Handling the mask
The following methods can be used to access information about the mask or to manipulate the mask.
__setmask__(mask, copy=False)
    Set the mask.

harden_mask()
    Force the mask to hard.

    Whether the mask of a masked array is hard or soft is determined by its
    hardmask property. harden_mask sets hardmask to True.

    See Also:
    hardmask

soften_mask()
    Force the mask to soft.

    Whether the mask of a masked array is hard or soft is determined by its
    hardmask property. soften_mask sets hardmask to False.

    See Also:
    hardmask

unshare_mask()
    Copy the mask and set the sharedmask flag to False.

    Whether the mask is shared between masked arrays can be seen from the
    sharedmask property. unshare_mask ensures the mask is not shared. A copy
    of the mask is only made if it was shared.

    See Also:
    sharedmask

shrink_mask()
    Reduce a mask to nomask when possible.

    Parameters
    None:

    Returns
    None:

    Examples

    >>> x = np.ma.array([[1, 2], [3, 4]], mask=[0]*4)
    >>> x.mask
    array([[False, False],
           [False, False]], dtype=bool)
    >>> x.shrink_mask()
    >>> x.mask
    False

Handling the fill_value

get_fill_value(self)
    Return the filling value of the masked array.

set_fill_value(self, value)
    Set the filling value of the masked array.
get_fill_value()  
Return the filling value of the masked array.

Returns  
fill_value : scalar
The filling value.

Examples

```python
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...   np.ma.array([0, 1], dtype=dt).get_fill_value()
...
999999
999999
1e+20
(1e+20+0j)
```

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.get_fill_value()
-inf
```

set_fill_value(value=None)
Set the filling value of the masked array.

Parameters  
value : scalar, optional
The new filling value. Default is None, in which case a default based on the data type
is used.

See Also:

ma.set_fill_value
Equivalent function.

Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.set_fill_value(np.pi)
>>> x.fill_value
3.1415926535897931
```

Reset to default:

```python
>>> x.set_fill_value()
>>> x.fill_value
1e+20
```

Counting the missing elements

MaskedArray.count(self, axis=None)
Count the non-masked elements of the array along the given axis.

count(axis=None)
Count the non-masked elements of the array along the given axis.
Parameters

axis : int, optional

Axis along which to count the non-masked elements. If axis is None, all non-masked elements are counted.

Returns

result : int or ndarray

If axis is None, an integer count is returned. When axis is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:

count_masked
Count masked elements in array or along a given axis.

Examples

>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(data =
[[0 1 2]
[-- -- --]],
mask =
[[False False False]
[ True  True  True]],
fill_value = 999999)

>>> a.count()
3

When the axis keyword is specified an array of appropriate size is returned.

>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])

1.6.7 Masked array operations

Constants

<table>
<thead>
<tr>
<th>ma.MaskType</th>
<th>Numpy’s Boolean type. Character code: ?. Alias: bool8</th>
</tr>
</thead>
</table>

MaskType
alias of bool_
Creation

From existing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.masked_array</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.array</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.copy</code></td>
<td>Copy a copy(order='C')</td>
</tr>
<tr>
<td><code>ma.frombuffer</code></td>
<td>Interpret a buffer as a 1-dimensional array.</td>
</tr>
<tr>
<td><code>ma.fromfunction</code></td>
<td>Construct an array by executing a function over each coordinate.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.copy</code></td>
<td>Return a copy of the array.</td>
</tr>
</tbody>
</table>

**masked_array**

Alias of `MaskedArray`

**array**

An array class with possibly masked values.

Masked values of True exclude the corresponding element from any computation.

Construction:

```python
x = MaskedArray(data, mask=nomask, dtype=None, copy=True,
                fill_value=None, keep_mask=True, hard_mask=False,
                shrink=True)
```

**Parameters**

- **data**: array_like
  - Input data.
- **mask**: sequence, optional
  - Mask. Must be convertible to an array of booleans with the same shape as `data`. True indicates a masked (i.e. invalid) data.
- **dtype**: dtype, optional
  - Data type of the output. If `dtype` is None, the type of the data argument (`data.dtype`) is used. If `dtype` is not None and different from `data.dtype`, a copy is performed.
- **copy**: bool, optional
  - Whether to copy the input data (True), or to use a reference instead. Default is False.
- **subok**: bool, optional
  - Whether to return a subclass of `MaskedArray` if possible (True) or a plain `MaskedArray`. Default is True.
- **ndmin**: int, optional
  - Minimum number of dimensions. Default is 0.
- **fill_value**: scalar, optional
  - Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.
- **keep_mask**: bool, optional
  - Whether to combine `mask` with the mask of the input data, if any (True), or to use only `mask` for the output (False). Default is True.
hard_mask : bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be un-
masked. Default is False.

shrink : bool, optional

Whether to force compression of an empty mask. Default is True.

copy()
copy a.copy(order='C')

Return a copy of the array.

Parameters

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

By default, the result is stored in C-contiguous (row-major) order in memory. If order is
F, the result has ‘Fortran’ (column-major) order. If order is ‘A’ ('Any'), then the result
has the same order as the input.

Examples

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

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

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

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

frombuffer()
Interpret a buffer as a 1-dimensional array.

Parameters

buffer :

An object that exposes the buffer interface.

dtype : data-type, optional

Data type of the returned array.

count : int, optional

Number of items to read. -1 means all data in the buffer.

offset : int, optional

Start reading the buffer from this offset.
Notes

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```python
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```python
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

fromfunction()

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value \( f(x, y, z) \) at coordinate \((x, y, z)\).

Parameters

- **function**: callable
  The function is called with N parameters, each of which represents the coordinates of the array varying along a specific axis. For example, if `shape` were \((2, 2)\), then the parameters would be two arrays, \([0, 0], [1, 1]\) and \([0, 1], [0, 1]\). Function must be capable of operating on arrays, and should return a scalar value.

- **shape**: (N,) tuple of ints
  Shape of the output array, which also determines the shape of the coordinate arrays passed to `function`.

- **dtype**: data-type, optional
  Data-type of the coordinate arrays passed to `function`. By default, `dtype` is float.

Returns

- **out**: any
  The result of the call to `function` is passed back directly. Therefore the type and shape of `out` is completely determined by `function`.

See Also:

- indices, meshgrid

Notes

Keywords other than `shape` and `dtype` are passed to `function`.

Examples

```python
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)
```

144 Chapter 1. Array objects
```python
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

copy()
Return a copy of the array.

Parameters

order : {'C', 'F', 'A'}, optional
By default, the result is stored in C-contiguous (row-major) order in memory. If order is 'F', the result has ‘Fortran’ (column-major) order. If order is ‘A’ (‘Any’), then the result has the same order as the input.

Examples

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

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

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

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

Ones and zeros

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.empty(shape, dtype, order)</td>
<td>Return a new array of given shape and type, without initializing entries.</td>
</tr>
<tr>
<td>ma.empty_like(a)</td>
<td>Return a new array with the same shape and type as a given array.</td>
</tr>
<tr>
<td>ma.masked_all(shape, dtype)</td>
<td>Empty masked array with all elements masked.</td>
</tr>
<tr>
<td>ma.masked_all_like(arr)</td>
<td>Empty masked array with the properties of an existing array.</td>
</tr>
<tr>
<td>ma.ones(shape, dtype, order)</td>
<td>Return a new array of given shape and type, filled with ones.</td>
</tr>
<tr>
<td>ma.zeros(shape, dtype, order)</td>
<td>Return a new array of given shape and type, filled with zeros.</td>
</tr>
</tbody>
</table>

empty()
Return a new array of given shape and type, without initializing entries.

Parameters

shape : int or tuple of int
Shape of the empty array
dtype : data-type, optional
Desired output data-type.
order : {'C', 'F'}, optional

1.6. Masked arrays
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

**See Also:**

`empty_like`, `zeros`, `ones`

**Notes**

`empty`, unlike `zeros`, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

**Examples**

```python
>>> np.empty([2, 2])
array([[ -9.74499359e+001,  6.69583040e-309],
      [ 2.13182611e-314,  3.06959433e-309]])
```

```python
>>> np.empty([2, 2], dtype=int)
array([[ 1073741821, -1067949133],
      [ 496041986, 19249760]])
```

`empty_like()`

Return a new array with the same shape and type as a given array.

**Parameters**

`a` : array_like

The shape and data-type of `a` define the parameters of the returned array.

**Returns**

`out` : ndarray

Array of random data with the same shape and type as `a`.

**See Also:**

`ones_like`

Return an array of ones with shape and type of input.

`zeros_like`

Return an array of zeros with shape and type of input.

`empty`

Return a new uninitialized array.

`ones`

Return a new array setting values to one.

`zeros`

Return a new array setting values to zero.

**Notes**

This function does *not* initialize the returned array; to do that use `zeros_like` or `ones_like` instead. It may be marginally faster than the functions that do set the array values.

**Examples**
>>> a = ([1,2,3], [4,5,6])
   # a is array-like
>>> np.empty_like(a)
array([[-1073741821, -1073741821,  3],
       [   0,           0, -1073741821]])
>>> a = np.array([[1., 2., 3.],[4.,5.,6.]])
>>> np.empty_like(a)
array([[ -2.00000715e+000,  1.48219694e-323, -2.00000572e+000],
       [ 4.38791518e-305, -2.00000715e+000,  4.17269252e-309]])

**masked_all** *(shape, dtype= '<type 'float'>')*

Empty masked array with all elements masked.

Return an empty masked array of the given shape and dtype, where all the data are masked.

**Parameters**

- **shape** : tuple
  Shape of the required MaskedArray.
- **dtype** : dtype, optional
  Data type of the output.

**Returns**

- **a** : MaskedArray
  A masked array with all data masked.

**See Also:**

**masked_all_like**

Empty masked array modelled on an existing array.

**Examples**

```python
>>> import numpy.ma as ma
>>> ma.masked_all((3, 3))
masked_array(data =
             [[-- -- --]
              [-- -- --]
              [-- -- --]],
       mask =
       [[ True  True  True]
        [ True  True  True]
        [ True  True  True]],
      fill_value=1e+20)
```

The **dtype** parameter defines the underlying data type.

```python
>>> a = ma.masked_all((3, 3))
>>> a.dtype
dtype('float64')
>>> a = ma.masked_all((3, 3), dtype=np.int32)
>>> a.dtype
dtype('int32')
```

**masked_all_like**(arr)

Empty masked array with the properties of an existing array.

Return an empty masked array of the same shape and dtype as the array **arr**, where all the data are masked.

1.6. Masked arrays
Parameters

arr : ndarray

An array describing the shape and dtype of the required MaskedArray.

Returns

a : MaskedArray

A masked array with all data masked.

Raises

AttributeError :

If arr doesn’t have a shape attribute (i.e. not an ndarray)

See Also:

masked_all

Empty masked array with all elements masked.

Examples

```python
>>> import numpy.ma as ma
>>> arr = np.zeros((2, 3), dtype=np.float32)
>>> arr
array([[ 0., 0., 0.],
       [ 0., 0., 0.]], dtype=float32)
>>> ma.masked_all_like(arr)
masked_array(data =
    [[-- -- --],
     [-- -- --]],
    mask =
    [[ True  True  True],
     [ True  True  True]],
    fill_value=1e+20)
```

The dtype of the masked array matches the dtype of arr.

```python
>>> arr.dtype
dtype('float32')
>>> ma.masked_all_like(arr).dtype
dtype('float32')
```

ones ()

Return a new array of given shape and type, filled with ones.

Please refer to the documentation for zeros.

See Also:

zeros

Examples

```python
>>> np.ones(5)
array([ 1., 1., 1., 1., 1.])
```

```python
>>> np.ones((5,), dtype=np.int)
array([1, 1, 1, 1, 1])
```
zeros()

Return a new array of given shape and type, filled with zeros.

Parameters

shape: int or sequence of ints
    Shape of the new array, e.g., (2, 3) or 2.

dtype: data-type, optional
    The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.

order: {'C', 'F'}, optional
    Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-
    wise) order in memory.

Returns

out: ndarray
    Array of zeros with the given shape, dtype, and order.

See Also:

zeros_like
    Return an array of zeros with shape and type of input.

ones_like
    Return an array of ones with shape and type of input.

empty_like
    Return an empty array with shape and type of input.

ones
    Return a new array setting values to one.

eempty
    Return a new uninitialized array.

Examples

>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])

>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')])  # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])

Inspecting the array

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.all(self[, axis, out])</td>
<td>Check if all of the elements of ( a ) are true.</td>
</tr>
<tr>
<td>ma.any(self[, axis, out])</td>
<td>Check if any of the elements of ( a ) are true.</td>
</tr>
<tr>
<td>ma.count(a[, axis])</td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td>ma.count_masked(arr[, axis])</td>
<td>Count the number of masked elements along the given axis.</td>
</tr>
<tr>
<td>ma.getmask(a)</td>
<td>Return the mask of a masked array, or nomask.</td>
</tr>
<tr>
<td>ma.getmaskarray(arr)</td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td>ma.getdata(a[, subok])</td>
<td>Return the data of a masked array as an ndarray.</td>
</tr>
<tr>
<td>ma.nonzero(self)</td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td>ma.shape(obj)</td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td>ma.size(obj[, axis])</td>
<td>Return the number of elements along a given axis.</td>
</tr>
<tr>
<td>ma.MaskedArray.all(self[, axis, out])</td>
<td>Check if all of the elements of ( a ) are true.</td>
</tr>
<tr>
<td>ma.MaskedArray.any(self[, axis, out])</td>
<td>Check if any of the elements of ( a ) are true.</td>
</tr>
<tr>
<td>ma.MaskedArray.count(self[, axis])</td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td>ma.MaskedArray.nonzero(self)</td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td>ma.MaskedArray.shape(obj)</td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td>ma.size(obj[, axis])</td>
<td>Return the number of elements along a given axis.</td>
</tr>
</tbody>
</table>

all()  
Check if all of the elements of \( a \) are true.  

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

Parameters
axis : {None, integer}
Axis to perform the operation over. If None, perform over flattened array.

out : {None, array}, optional
Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

call  
equivalent function
### Examples

```python
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

**any()**

Check if any of the elements of `a` are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

**Parameters**

- `axis` : {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array and return a scalar.

- `out` : {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

- `any` equivalent function

**count** (`a, axis=None`)

Count the non-masked elements of the array along the given axis.

**Parameters**

- `axis` : int, optional
  
  Axis along which to count the non-masked elements. If `axis` is `None`, all non-masked elements are counted.

**Returns**

- `result` : int or ndarray
  
  If `axis` is `None`, an integer count is returned. When `axis` is not `None`, an array with shape determined by the lengths of the remaining axes, is returned.

**See Also:**

- `count_masked` Count masked elements in array or along a given axis.

### Examples

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(data =
[[0 1 2]
[-- -- --]],
mask =
```

#### 1.6. Masked arrays
[False False False]
[ True True True],
    fill_value = 999999)
>>> a.count()
3

When the axis keyword is specified an array of appropriate size is returned.

>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])

count_masked(arr, axis=None)
Count the number of masked elements along the given axis.

Parameters
arr : array_like
    An array with (possibly) masked elements.
axis : int, optional
    Axis along which to count. If None (default), a flattened version of the array is used.

Returns
count : int, ndarray
    The total number of masked elements (axis=None) or the number of masked elements
    along each slice of the given axis.

See Also:
MaskedArray.count
Count non-masked elements.

Examples
>>> import numpy.ma as ma
>>> a = np.arange(9).reshape((3,3))
>>> a = ma.array(a)
>>> a[1, 0] = ma.masked
>>> a[1, 2] = ma.masked
>>> a[2, 1] = ma.masked
>>> a
masked_array(data =
[[0 1 2]
[-- 4 --]
[6 -- 8]],
    mask =
[[False False False]
[ True False True]
[False True False]],
    fill_value=999999)
>>> ma.count_masked(a)
3

When the axis keyword is used an array is returned.
getmask(a)
Return the mask of a masked array, or nomask.

Return the mask of a as an ndarray if a is a MaskedArray and the mask is not nomask, else return nomask. To guarantee a full array of booleans of the same shape as a, use getmaskarray.

Parameters
   a : array_like
      Input MaskedArray for which the mask is required.

See Also:

gedata
Return the data of a masked array as an ndarray.

getmaskarray
Return the mask of a masked array, or full array of False.

Examples

>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1, 2], [3, 4]], 2)
>>> a
masked_array(data =
[[1 --]
 [3 4]],
 mask =
[[False  True]
 [False False]],
 fill_value=999999)
>>> ma.getmask(a)
array([[False, True],
       [False, False]], dtype=bool)

Equivalently use the MaskedArray mask attribute.

>>> a.mask
array([[False,  True],
       [False, False]], dtype=bool)

Result when mask == nomask

>>> b = ma.masked_array([[1, 2], [3, 4]])
>>> b
masked_array(data =
[[1 2]
 [3 4]],
 mask =
False,
 fill_value=999999)
>>> ma.nomask

1.6. Masked arrays
False
>>> ma.getmask(b) == ma.nomask
True
>>> b.mask == ma.nomask
True

getmaskarray(arr)
Return the mask of a masked array, or full boolean array of False.

Return the mask of arr as an ndarray if arr is a MaskedArray and the mask is not nomask, else return a full boolean array of False of the same shape as arr.

Parameters
arr : array_like
    Input MaskedArray for which the mask is required.

See Also:

getmask
    Return the mask of a masked array, or nomask.

getdata
    Return the data of a masked array as an ndarray.

Examples

>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False True]
[False False]],
fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
[False, False]], dtype=bool)

Result when mask == nomask

>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
[3 4]],
mask =
False,
fill_value=999999)
>>> ma.getmaskarray(b)
array([[False, False],
[False, False]], dtype=bool)

gedata(a, subok=True)
Return the data of a masked array as an ndarray.
Return the data of \( a \) (if any) as an ndarray if \( a \) is a \texttt{MaskedArray}, else return \( a \) as a ndarray or subclass (depending on \texttt{subok}) if not.

**Parameters**

\( a \) : array_like

Input \texttt{MaskedArray}, alternatively a ndarray or a subclass thereof.

\texttt{subok} : bool

Whether to force the output to be a *pure* ndarray (False) or to return a subclass of ndarray if appropriate (True, default).

**See Also:**

\texttt{getmask}

Return the mask of a masked array, or nomask.

\texttt{getmaskarray}

Return the mask of a masked array, or full array of False.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1, 2], [3, 4]], 2)
>>> a
masked_array(data =
[[1 --]
 [3 4]],
mask =
[[False True]
[False False]],
fill_value=999999)
>>> ma.getdata(a)
array([[1, 2],
 [3, 4]])
```

Equivalently use the \texttt{MaskedArray data} attribute.

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

**nonzero()**

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

\( a[a.nonzero()] \)

To group the indices by element, rather than dimension, use instead:

\( \text{np.transpose(a.nonzero())} \)

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

None :
Returns
tuple_of_arrays : tuple
   Indices of elements that are non-zero.

See Also:
numpy.nonzero
   Function operating on ndarrays.
flatnonzero
   Return indices that are non-zero in the flattened version of the input array.
ndarray.nonzero
   Equivalent ndarray method.

Examples

```python
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]],
   mask =
   fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))
```
Masked elements are ignored.

```python
>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[1.0 0.0 0.0]
 [0.0 -- 0.0]
 [0.0 0.0 1.0]],
   mask =
[[False False False]
 [False  True False]
 [False False False]],
   fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
```
Indices can also be grouped by element.

```python
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `ma.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

156 Chapter 1. Array objects
>>> a = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
masked_array(data =
[[False False False]
 [ True  True  True]
 [ True  True  True]],
   mask =
False,
   fill_value=999999)

>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

The nonzero method of the condition array can also be called.

>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

**shape** *(obj)*
Return the shape of an array.

Parameters
---

  a : array_like

    Input array.

Returns
---

  shape : tuple of ints

    The elements of the shape tuple give the lengths of the corresponding array dimensions.

See Also:
---

  alen

**ndarray.shape**
Equivalent array method.

Examples
---

```python
>>> np.shape(np.eye(3))
(3, 3)

```

```python
>>> np.shape([[1, 2]])
(1, 2)

```

```python
>>> np.shape([0])
(1,)

```

```python
>>> np.shape(0)
()  # empty tuple

```

```python
>>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)

```

```python
>>> a.shape
(2,)

```

**size** *(obj, axis=None)*
Return the number of elements along a given axis.

Parameters
---

  a : array_like

Examples
---

```python
>>> a = np.array([[1, 2], (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)

```

```python
>>> a.size
(2,)

```

1.6. Masked arrays
Input data.

**axis**: int, optional

Axis along which the elements are counted. By default, give the total number of elements.

**Returns**

**element_count**: int

Number of elements along the specified axis.

**See Also**:

**shape**

dimensions of array

**ndarray.shape**

dimensions of array

**ndarray.size**

number of elements in array

**Examples**

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.size(a)
6
>>> np.size(a, 1)
3
>>> np.size(a, 0)
2
```

**data**

Return the current data, as a view of the original underlying data.

**mask**

Mask

**recordmask**

Return the mask of the records. A record is masked when all the fields are masked.

**all** *(axis=None, out=None)*

Check if all of the elements of *a* are true.

Performs a **logical_and** over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**

- **axis**: {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array.

- **out**: {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also**:

**all**

equivalent function
Examples

```python
>>> np.ma.array([1, 2, 3]).all()
True
>>> a = np.ma.array([1, 2, 3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

any (axis=None, out=None)

Check if any of the elements of `a` are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

Parameters

- `axis` : {None, integer}
  - Axis to perform the operation over. If None, perform over flattened array and return a scalar.
- `out` : {None, array}, optional
  - Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

- any
- equivalent function

count (axis=None)

Count the non-masked elements of the array along the given axis.

Parameters

- `axis` : int, optional
  - Axis along which to count the non-masked elements. If `axis` is None, all non-masked elements are counted.

Returns

- `result` : int or ndarray
  - If `axis` is None, an integer count is returned. When `axis` is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:

- count_masked
- Count masked elements in array or along a given axis.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(data =
  [[ 0  1  2]
   [-- -- --]],
  mask =
```
When the `axis` keyword is specified an array of appropriate size is returned.

```python
>>> a.count(axis=0)
array([1, 1, 1])

>>> a.count(axis=1)
array([3, 0])
```

nonzero()

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```python
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use instead:

```python
np.transpose(a.nonzero())
```

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

None:

**Returns**

**tuple_of_arrays**: tuple

Indices of elements that are non-zero.

See Also:

`numpy.nonzero`

Function operating on ndarrays.

`flatnonzero`

Return indices that are non-zero in the flattened version of the input array.

`ndarray.nonzero`

Equivalent ndarray method.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))

masked_array(data =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]],
mask =
False,
fill_value=1e+20)
```
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[1.0 0.0 0.0]
[0.0 -- 0.0]
[0.0 0.0 1.0]],
mask =
[[False False False]
[False True False]
[False False False]],
fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))

Indices can also be grouped by element.

>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array $a$, the condition $a > 3$ is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the $a$ where the condition is true.

>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(data =
[[False False False]
[ True True True]
[ True True True]],
mask =
False,
fill_value=999999)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

The nonzero method of the condition array can also be called.

>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

**shape** *(obj)*  
Return the shape of an array.

Parameters

- **a**: array_like

  Input array.

Returns

- **shape**: tuple of ints

  The elements of the shape tuple give the lengths of the corresponding array dimensions.
See Also:

alen

ndarray.shape
   Equivalent array method.

Examples

>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()

>>> a = np.array([[1, 2, 3], [4, 5, 6]], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)

size (obj, axis=None)
   Return the number of elements along a given axis.

Parameters
   a : array_like
      Input data.

axis : int, optional
      Axis along which the elements are counted. By default, give the total number of elements.

Returns
   element_count : int
      Number of elements along the specified axis.

See Also:

shape
dimensions of array

ndarray.shape
dimensions of array

ndarray.size
   number of elements in array

Examples

>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.size(a)
6
>>> np.size(a, 1)
3
Manipulating a MaskedArray

Changing the shape

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.ravel(self)</code></td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td><code>ma.reshape(a, new_shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>ma.resize(x, new_shape)</code></td>
<td>Return a new masked array with the specified size and shape.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.flatten([order])</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.ravel(self)</code></td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.reshape(self, *s, **kwargs)</code></td>
<td>Give a new shape to the array without changing its data.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.resize(self, newshape[, resize, order])</code></td>
<td>Return a new masked array with the specified size and shape.</td>
</tr>
</tbody>
</table>

**ravel()**

Returns a 1D version of self, as a view.

Returns

**MaskedArray**

Output view is of shape `(self.size,)` (or `(np.ma.product(self.shape),,)`).

**Examples**

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

**reshape(a, new_shape, order='C')**

Returns an array containing the same data with a new shape.

Refer to `MaskedArray.reshape` for full documentation.

See Also:

- `MaskedArray.reshape`
  equivalent function

**resize(x, new_shape)**

Return a new masked array with the specified size and shape.

This is the masked equivalent of the `numpy.resize` function. The new array is filled with repeated copies of `x` (in the order that the data are stored in memory). If `x` is masked, the new array will be masked, and the new mask will be a repetition of the old one.
See Also:

numpy.resize
   Equivalent function in the top level NumPy module.

Examples

>>> import numpy.ma as ma
>>> a = ma.array([[1, 2], [3, 4]])
>>> a[0, 1] = ma.masked
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False  True]
[False False]],
fill_value = 999999)

>>> np.resize(a, (3, 3))
array([[1, 2, 3],
[4, 1, 2],
[3, 4, 1]])

>>> ma.resize(a, (3, 3))
masked_array(data =
[[1 -- 3]
[4 1 --]
[3 4 1]],
mask =
[[False  True False]
[False False True]
[False False False]],
fill_value = 999999)

A MaskedArray is always returned, regardless of the input type.

>>> a = np.array([[1, 2], [3, 4]])
>>> ma.resize(a, (3, 3))
masked_array(data =
[[1 2 3]
[4 1 2]
[3 4 1]],
mask =
False,
fill_value = 999999)

flatten()
Return a copy of the array collapsed into one dimension.

Parameters
   order : {'C', 'F'}, optional
      Whether to flatten in C (row-major) or Fortran (column-major) order. 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])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

**ravel()**

Returns a 1D version of self, as a view.

**Examples**

```python
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
[7 -- 9]]
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

**reshape (*s, **kwargs)**

Give a new shape to the array without changing its data.

Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

**Parameters**

- **shape**: int or tuple of ints
  The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

- **order**: {'C', 'F'}, optional
  Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

**Returns**

- **reshaped_array**: array
  A new view on the array.

See Also:

**reshape**

Equivalent function in the masked array module.
**numpy.ndarray.reshape**
Equivalent method on ndarray object.

**numpy.reshape**
Equivalent function in the NumPy module.

**Notes**
The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`

**Examples**

```python
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
>>> print x
[[- 2]
 [3 --]]

>>> x = x.reshape((4,1))
>>> print x
[[-]
 [2]
 [3]
[--]]
```

**resize** *(newshape, refcheck=True, order=False)*

**Warning:** This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the `numpy.ma.resize` function instead.

This method is difficult to implement safely and may be deprecated in future releases of NumPy.

**Modifying axes**

| `ma.swapaxes()` | `swapaxes a.swapaxes(axis1, axis2)` |
| `ma.transpose(a, axes)` | `permute the dimensions of an array.` |
| `ma.MaskedArray.swapaxes(axis1, axis2)` | `Return a view of the array with axis1 and axis2 interchanged.` |
| `ma.MaskedArray.transpose(*axes)` | `Returns a view of the array with axes transposed.` |

**swapaxes()**

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

**See Also:**

- `numpy.swapaxes`
  - equivalent function

**transpose(a, axes=None)**

- `permute the dimensions of an array.`
  - This function is exactly equivalent to `numpy.transpose`.

**See Also:**

- `numpy.transpose`
  - Equivalent function in top-level NumPy module.
Examples

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked

>>> x
masked_array(data =
[[0 1]
[2 --]],
mask =
[[False False]
[False  True]],
fill_value = 999999)

>>> ma.transpose(x)
masked_array(data =
[[0 2]
[1 --]],
mask =
[[False False]
[False  True]],
fill_value = 999999)
```

**swapaxes()**

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See Also:**

`numpy.swapaxes`

equivalent function

**transpose()**

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]])
```

Changing the number of dimensions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atleast_1d(*arys)</code></td>
<td>Convert inputs to arrays with at least one dimension. Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.</td>
</tr>
<tr>
<td><code>atleast_2d(*arys)</code></td>
<td>View inputs as arrays with at least two dimensions.</td>
</tr>
<tr>
<td><code>atleast_3d(*arys)</code></td>
<td>View inputs as arrays with at least three dimensions.</td>
</tr>
<tr>
<td><code>expand_dims(x, axis)</code></td>
<td>Expand the shape of an array.</td>
</tr>
<tr>
<td><code>squeeze(a)</code></td>
<td>Remove single-dimensional entries from the shape of an array.</td>
</tr>
<tr>
<td><code>squeeze()</code></td>
<td>Remove single-dimensional entries from the shape of <code>a</code>.</td>
</tr>
<tr>
<td><code>column_stack(tup)</code></td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td><code>concatenate(arrays[, axis])</code></td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
<tr>
<td><code>dstack(tup)</code></td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td><code>hstack(tup)</code></td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td><code>hsplit(ary, indices_or_sections)</code></td>
<td>Split an array into multiple sub-arrays horizontally (column-wise).</td>
</tr>
<tr>
<td><code>mr_</code></td>
<td>Translate slice objects to concatenation along the first axis.</td>
</tr>
<tr>
<td><code>row_stack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
<tr>
<td><code>vstack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

### `atleast_1d(*arys)`

Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**

- `array1, array2, ...` : array_like

  One or more input arrays.
Returns

ret : ndarray

An array, or sequence of arrays, each with `a.ndim >= 1`. Copies are made only if necessary.

See Also:

`atleast_2d`, `atleast_3d`

Examples

```python
>>> np.atleast_1d(1.0)
array([ 1.])

>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[ 0., 1., 2.],
      [ 3., 4., 5.],
      [ 6., 7., 8.]])
>>> np.atleast_1d(x) is x
True

>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]
```

`atleast_2d(*arys)`

View inputs as arrays with at least two dimensions.

Parameters

array1, array2, ... : array_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

Returns

res, res2, ... : ndarray

An array, or tuple of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

See Also:

`atleast_1d`, `atleast_3d`

Examples

```python
>>> np.atleast_2d(3.0)
array([[ 3.]])

>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[ 0., 1., 2.]])
>>> np.atleast_2d(x).base is x
True

>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([1]), array([1, 2]), array([[1, 2]])]
```
**atleast_3d**(*arys*)

View inputs as arrays with at least three dimensions.

**Parameters**

array1, array2, ... : array_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

**Returns**

res1, res2, ... : ndarray

An array, or tuple of arrays, each with :math:`a._\text{ndim} \geq 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape :math:`N` becomes a view of shape :math:`(1, N, 1)`. A 2-D array of shape :math:`(M, N)` becomes a view of shape :math:`(M, N, 1)`.

**See Also:**

atleast_1d, atleast_2d

**Examples**

```python
gnp.atleast_3d(3.0)
array([[ 3.]])
```

```python
g = np.arange(3.0)
gnp.atleast_3d(g).shape
(1, 3, 1)
```

```python
g = np.arange(12.0).reshape(4,3)
gnp.atleast_3d(g).shape
(4, 3, 1)
gnp.atleast_3d(g).base is g
True
```

```python
gfor arr in gnp.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
g... print arr, arr.shape
...
[[[1]
 [2]]] (1, 2, 1)
[[[1]
 [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)
```

**expand_dims**(*x, axis*)

Expand the shape of an array.

Expands the shape of the array by including a new axis before the one specified by the `axis` parameter. This function behaves the same as `numpy.expand_dims` but preserves masked elements.

**See Also:**

`numpy.expand_dims`

Equivalent function in top-level NumPy module.
Examples

```python
>>> import numpy.ma as ma
>>> x = ma.array([1, 2, 4])
>>> x[1] = ma.masked
>>> x
masked_array(data = [1 -- 4],
             mask = [False True False],
             fill_value = 999999)
>>> np.expand_dims(x, axis=0)
array([[1, 2, 4]])
>>> ma.expand_dims(x, axis=0)
masked_array(data =
             [[1 -- 4]],
             mask =
             [[False True False]],
             fill_value = 999999)
```

The same result can be achieved using slicing syntax with `np.newaxis`.

```python
>>> x[np.newaxis, :]
masked_array(data =
             [[1 -- 4]],
             mask =
             [[False True False]],
             fill_value = 999999)
```

`squeeze` *(a)*

Remove single-dimensional entries from the shape of an array.

**Parameters**

- `a`: array_like
  
  Input data.

**Returns**

- `squeezed`: ndarray

The input array, but with with all dimensions of length 1 removed. Whenever possible, a view on *a* is returned.

**Examples**

```python
>>> x = np.array([[0], [1], [2]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
```

`squeeze` ()

Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

See Also:

- `numpy.squeeze` equivalent function
column_stack()

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are
stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters
tup : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

Returns
stacked : 2-D array

The array formed by stacking the given arrays.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

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

concatenate(arrays, axis=0)

Concatenate a sequence of arrays along the given axis.

Parameters
arrays : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to axis
(the first, by default).

axis : int, optional

The axis along which the arrays will be joined. Default is 0.

Returns
result : MaskedArray

The concatenated array with any masked entries preserved.

See Also:

numpy.concatenate
Equivalent function in the top-level NumPy module.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
```
```python
masked_array(data = [0 -- 2],
    mask = [False  True False],
    fill_value = 999999)
>>>
b
masked_array(data = [2 3 4],
    mask = False,
    fill_value = 999999)
>>>
ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
    mask = [False  True False False False False],
    fill_value = 999999)
```

**dstack()**

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by `dsplit`. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

**Parameters**

- `tup`: sequence of arrays

  Arrays to stack. All of them must have the same shape along all but the third axis.

**Returns**

- `stacked`: ndarray

  The array formed by stacking the given arrays.

**See Also:**

- `vstack`
  Stack along first axis.

- `hstack`
  Stack along second axis.

- `concatenate`
  Join arrays.

- `dsplit`
  Split array along third axis.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

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

**hstack()**

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

**Parameters**

- **tup**: sequence of ndarrays

  All arrays must have the same shape along all but the second axis.

**Returns**

- **stacked**: ndarray

  The array formed by stacking the given arrays.

**See Also:**

- **vstack**
  Stack arrays in sequence vertically (row wise).
- **dstack**
  Stack arrays in sequence depth wise (along third axis).
- **concatenate**
  Join a sequence of arrays together.
- **hsplit**
  Split array along second axis.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

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

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a, b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```
Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the split documentation. hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.

See Also:

split
Split an array into multiple sub-arrays of equal size.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

```python
g = np.arange(16.0).reshape(4, 4)
g
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
>>> np.hsplit(g, 2)
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
 array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]])]

>>> np.hsplit(g, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
 array([[ 3.],
        [ 7.],
        [11.],
        [15.]],
        dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.

```python
g = np.arange(8.0).reshape(2, 2, 2)
g
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
>>> np.hsplit(g, 2)
[array([[[ 0.,  1.]],
        [[ 4.,  5.]]]),
 array([[[ 2.,  3.]],
        [[ 6.,  7.]]])]
```

mr_()
Translate slice objects to concatenation along the first axis.

1.6. Masked arrays
This is the masked array version of `lib.index_tricks.RClass`.

**See Also:**

`lib.index_tricks.RClass`

**Examples**

```python
gnp.ma.mr_([np.ma.array([1,2,3]), 0, 0, np.ma.array([4,5,6])]
array([1, 2, 3, 0, 0, 4, 5, 6])
```

`row_stack()`

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by `vsplit`.

**Parameters**

- `tup`: sequence of ndarrays

  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

**Returns**

- `stacked`: ndarray

  The array formed by stacking the given arrays.

**See Also:**

- `hstack`:

  Stack arrays in sequence horizontally (column wise).

- `dstack`:

  Stack arrays in sequence depth wise (along third dimension).

- `concatenate`:

  Join a sequence of arrays together.

- `vsplit`:

  Split array into a list of multiple sub-arrays vertically.

**Notes**

The function is applied to both the `_data` and the `_mask`, if any.

**Examples**

```python
gnp.a = np.array([1, 2, 3])
gnp.b = np.array([2, 3, 4])
gnp.vstack((a,b))
array([[[1, 2, 3],
       [2, 3, 4]])
```

```python
gnp.a = np.array([[1], [2], [3]])
gnp.b = np.array([[2], [3], [4]])
gnp.vstack((a,b))
array([[1],
       [2],
       [3],
       [4]])
```
vstack()

Stack arrays in sequence vertically (row wise).
Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

Parameters

tup : sequence of ndarrays
Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns

stacked : ndarray
The array formed by stacking the given arrays.

See Also:

hstack
Stack arrays in sequence horizontally (column wise).
dstack
Stack arrays in sequence depth wise (along third dimension).
concatenate
Join a sequence of arrays together.
vsplit
Split array into a list of multiple sub-arrays vertically.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

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

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```
Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.column_stack(tup)</code></td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td><code>ma.concatenate(arrays[, axis])</code></td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
<tr>
<td><code>ma.dstack(tup)</code></td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td><code>ma.hstack(tup)</code></td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td><code>ma.vstack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

**column_stack()**

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

**Parameters**

- `tup`: sequence of 1-D or 2-D arrays.
  
  Arrays to stack. All of them must have the same first dimension.

**Returns**

- `stacked`: 2-D array
  
  The array formed by stacking the given arrays.

**Notes**

The function is applied to both the `_data` and the `_mask`, if any.

**Examples**

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

**concatenate (arrays, axis=0)**

Concatenate a sequence of arrays along the given axis.

**Parameters**

- `arrays`: sequence of array_like
The arrays must have the same shape, except in the dimension corresponding to \textit{axis} (the first, by default).

\textbf{axis} : int, optional

The axis along which the arrays will be joined. Default is 0.

\textbf{Returns}

\textbf{result} : MaskedArray

The concatenated array with any masked entries preserved.

\textbf{See Also:}

\textbf{numpy.concatenate}

Equivalent function in the top-level NumPy module.

\textbf{Examples}

\begin{verbatim}
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
            fill_value = 999999)
>>> b
masked_array(data = [2 3 4],
             mask = False,
            fill_value = 999999)
>>> ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
            fill_value = 999999)
\end{verbatim}

\textbf{dstack}()

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by \textbf{dsplit}. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

\textbf{Parameters}

\textbf{tup} : sequence of arrays

Arrays to stack. All of them must have the same shape along all but the third axis.

\textbf{Returns}

\textbf{stacked} : ndarray

The array formed by stacking the given arrays.

\textbf{See Also:}

\textbf{vstack}

Stack along first axis.
**hstack**
Stack along second axis.

**concatenate**
Join arrays.

**dsplit**
Split array along third axis.

**Notes**
The function is applied to both the _data and the _mask, if any.

**Examples**

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

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

**hstack()**
Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

**Parameters**
- **tup**: sequence of ndarrays
  All arrays must have the same shape along all but the second axis.

**Returns**
- **stacked**: ndarray
  The array formed by stacking the given arrays.

**See Also:**

**vstack**
Stack arrays in sequence vertically (row wise).

**dstack**
Stack arrays in sequence depth wise (along third axis).

**concatenate**
Join a sequence of arrays together.

**hsplit**
Split array along second axis.
Notes

The function is applied to both the _data and the _mask, if any.

Examples

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

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

`vstack()`

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by `vsplit`.

Parameters

- `tup` : sequence of ndarrays
  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns

- `stacked` : ndarray
  The array formed by stacking the given arrays.

See Also:

- `hstack`
  Stack arrays in sequence horizontally (column wise).
- `dstack`
  Stack arrays in sequence depth wise (along third dimension).
- `concatenate`
  Join a sequence of arrays together.
- `vsplit`
  Split array into a list of multiple sub-arrays vertically.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

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

**1.6. Masked arrays**
array([[1, 2, 3],
       [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])

Operations on masks

Creating a mask

Create a boolean mask from an array.

\[ \text{ma.make_mask} \quad (m[, \text{copy, shrink, flag} , ...]) \]

Create a boolean mask from an array.

Return \( m \) as a boolean mask, creating a copy if necessary or requested. The function can accept any sequence that is convertible to integers, or \texttt{nomask}. Does not require that contents must be 0s and 1s, values of 0 are interpreted as False, everything else as True.

**Parameters**

- **m** : array_like
  
  Potential mask.

- **copy** : bool, optional
  
  Whether to return a copy of \( m \) (True) or \( m \) itself (False).

- **shrink** : bool, optional
  
  Whether to shrink \( m \) to \texttt{nomask} if all its values are False.

- **flag** : bool, optional
  
  Deprecated equivalent of \texttt{shrink}.

- **dtype** : dtype, optional
  
  Data-type of the output mask. By default, the output mask has a dtype of \texttt{MaskType} (bool). If the dtype is flexible, each field has a boolean dtype.

**Returns**

- **result** : ndarray
  
  A boolean mask derived from \( m \).
Examples

```python
>>> import numpy.ma as ma
>>> m = [True, False, True, True]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
>>> m = [1, 0, 1, 1]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
>>> m = [1, 0, 2, -3]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
```

Effect of the `shrink` parameter.

```python
>>> m = np.zeros(4)
>>> m
array([ 0., 0., 0., 0.])
>>> ma.make_mask(m)
False
>>> ma.make_mask(m, shrink=False)
array([False, False, False, False], dtype=bool)
```

Using a flexible `dtype`.

```python
>>> m = [1, 0, 1, 1]
>>> n = [0, 1, 0, 0]
>>> arr = []
... for man, mouse in zip(m, n):
...     arr.append((man, mouse))
>>> arr
[(1, 0), (0, 1), (1, 0), (1, 0)]
>>> dtype = np.dtype({'names': ['man', 'mouse'],
                   'formats': [np.int, np.int]})
>>> arr = np.array(arr, dtype=dtype)
>>> arr
array([(1, 0), (0, 1), (1, 0), (1, 0)],
      dtype=[(‘man’, ’<i4’), (‘mouse’, ’<i4’)])
>>> ma.make_mask(arr, dtype=dtype)
array([(True, False), (False, True), (True, False), (True, False)],
      dtype=[(‘man’, ’|b1’), (‘mouse’, ’|b1’)])
```

`make_mask_none (newshape, dtype=None)`

Return a boolean mask of the given shape, filled with False.

This function returns a boolean ndarray with all entries False, that can be used in common mask manipulations. If a complex dtype is specified, the type of each field is converted to a boolean type.

Parameters

- `newshape` : tuple
  A tuple indicating the shape of the mask.

- `dtype` : [None, dtype], optional
  If None, use a MaskType instance. Otherwise, use a new datatype with the same fields as `dtype`, converted to boolean types.

Returns

- `result` : ndarray

1.6. Masked arrays 183
An ndarray of appropriate shape and dtype, filled with False.

See Also:

make_mask
Create a boolean mask from an array.

make_mask_descr
Construct a dtype description list from a given dtype.

Examples

```python
>>> import numpy.ma as ma
>>> ma.make_mask_none((3,))
array([False, False, False], dtype=bool)
```

Defining a more complex dtype.

```python
>>> dtype = np.dtype({'names': ['foo', 'bar'],
                   'formats': [np.float32, np.int]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i4')])
>>> ma.make_mask_none((3,), dtype=dtype)
array([(False, False), (False, False), (False, False)],
      dtype=[('foo', '|b1'), ('bar', '|b1')])
```

mask_or (m1, m2, copy=False, shrink=True)
Combine two masks with the logical_or operator.

The result may be a view on m1 or m2 if the other is nomask (i.e. False).

Parameters

m1, m2 : array_like
Input masks.

copy : bool, optional
If copy is False and one of the inputs is nomask, return a view of the other input mask. Defaults to False.

shrink : bool, optional
Whether to shrink the output to nomask if all its values are False. Defaults to True.

Returns

mask : output mask
The result masks values that are masked in either m1 or m2.

Raises

ValueError:
If m1 and m2 have different flexible dtypes.

Examples

```python
>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([ True,  True,  True, False], dtype=bool)
```
make_mask_descr (ndtype)
Construct a dtype description list from a given dtype.

Returns a new dtype object, with the type of all fields in ndtype to a boolean type. Field names are not altered.

Parameters
   ndtype : dtype
       The dtype to convert.

Returns
   result : dtype
       A dtype that looks like ndtype, the type of all fields is boolean.

Examples
>>> import numpy.ma as ma
>>> dtype = np.dtype([('names':[`foo`, `bar`],
       `formats':[np.float32, np.int])
>>> dtype
dtype([(`foo`, '<f4'), (`bar`, '<i4')])
>>> ma.make_mask_descr(dtype)
dtype([(`foo`, '|b1'), (`bar`, '|b1'))]
>>> ma.make_mask_descr(np.float32)
<type 'numpy.bool_'>

Accessing a mask
ma.getmask (a) Return the mask of a masked array, or nomask.
ma.getmaskarray (arr) Return the mask of a masked array, or full boolean array of False.
ma.masked_array.mask Mask

getmask (a)
Return the mask of a masked array, or nomask.

Return the mask of a as an ndarray if a is a MaskedArray and the mask is not nomask, else return nomask. To guarantee a full array of booleans of the same shape as a, use getmaskarray.

Parameters
   a : array_like
       Input MaskedArray for which the mask is required.

See Also:
getdata
Return the data of a masked array as an ndarray.

getmaskarray
Return the mask of a masked array, or full array of False.

Examples
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
 [3 4]],

1.6. Masked arrays
```
mask =
[[False  True],
 [False False]],
fill_value=999999)

>>> ma.getmask(a)
array([[False,  True],
       [False, False]], dtype=bool)
```

Equivalently use the `MaskedArray` mask attribute.

```
>>> a.mask
array([[False,  True],
       [False, False]], dtype=bool)
```

Result when mask == `nomask`

```
>>> b = ma.masked_array([[1,2],[3,4]])

>>> b
masked_array(data =
[[1 2],
 [3 4]],
mask = False,
fill_value=999999)

>>> ma.nomask
False

>>> ma.getmask(b) == ma.nomask
True

>>> b.mask == ma.nomask
True
```

`getmaskarray` *(arr)*

Return the mask of a masked array, or full boolean array of False.

Return the mask of `arr` as an ndarray if `arr` is a `MaskedArray` and the mask is not `nomask`, else return a full boolean array of False of the same shape as `arr`.

**Parameters**

`arr` : array_like

Input `MaskedArray` for which the mask is required.

**See Also:**

`getmask`

Return the mask of a masked array, or nomask.

`getdata`

Return the data of a masked array as an ndarray.

**Examples**

```
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --],
 [3 4]],
mask =
[[False  True],
 [False False]],
fill_value=999999)
```
mask =
[[False, True],
 [False, False]],
fill_value=999999)

>>> ma.getmaskarray(a)
array([[False,  True],
       [False, False]], dtype=bool)

Result when mask == nomask

>>> b = ma.masked_array([[1, 2], [3, 4]])
>>> b
masked_array(data =
[[1 2]
 [3 4]],
mask =
False,
fill_value=999999)

>>> >ma.getmaskarray(b)
array([[False, False],
       [False, False]], dtype=bool)

mask
Mask

Finding masked data

.. currentmodule:: numpy.ma

.. autosummary::
   :toctree: generated/

   flatnotmasked_contiguous
   flatnotmasked_edges
   notmasked_contiguous
   notmasked_edges

flatnotmasked_contiguous

Find contiguous unmasked data in a masked array along the given axis.

Parameters

- a : array
  The input array.

Returns

- slice_list : list
  A sorted sequence of slices (start index, end index).

See Also:

flatnotmasked_edges, notmasked_contiguous, notmasked_edges

Notes

Only accepts 2-D arrays at most.

Examples

```python
>>> a = np.arange(10)
>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> ma = np.ma.array(a, mask=mask)
>>> np.array(ma[~ma.mask])
array([3, 4, 6, 7, 8])
```
```python
>>> np.ma.extras.flatnotmasked_contiguous(ma)
[slice(3, 4, None), slice(6, 8, None)]
>>> ma = np.ma.array(a, mask=np.ones_like(a))
>>> print np.ma.extras.flatnotmasked_edges(ma)
None
```

**flatnotmasked_edges** *(a)*

Find the indices of the first and last unmasked values.

Expects a 1-D *MaskedArray*, returns None if all values are masked.

**Parameters**

- **arr**: array_like
  
  Input 1-D *MaskedArray*

**Returns**

- **edges**: ndarray or None
  
  The indices of first and last non-masked value in the array. Returns None if all values are masked.

**See Also**:

- flatnotmasked_contiguous, notmasked_contiguous, notmasked_edges

**Notes**

Only accepts 1-D arrays.

**Examples**

```python
>>> a = np.arange(10)
>>> m = (a < 3) | (a > 8) | (a == 5)

>>> ma = np.ma.array(a, mask=m)
>>> np.array(ma[~ma.mask])
array([3, 4, 6, 7, 8])

>>> flatnotmasked_edges(ma)
array([3, 8])

>>> ma = np.ma.array(a, mask=np.ones_like(a))
>>> print flatnotmasked_edges(ma)
None
```

**notmasked_contiguous** *(a, axis=None)*

Find contiguous unmasked data in a masked array along the given axis.

**Parameters**

- **a**: array_like
  
  The input array.

- **axis**: int, optional
  
  Axis along which to perform the operation. If None (default), applies to a flattened version of the array.
Returns

endpoints : list

A list of slices (start and end indexes) of unmasked indexes in the array.

See Also:

flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges

Notes

Only accepts 2-D arrays at most.

Examples

>>> a = np.arange(9).reshape((3, 3))
>>> mask = np.zeros_like(a)
>>> mask[1:, 1:] = 1

>>> ma = np.ma.array(a, mask=mask)
>>> np.array(ma[~ma.mask])
array([0, 1, 2, 3, 6])

>>> np.ma.extras.notmasked_contiguous(ma)
[slice(0, 3, None), slice(6, 6, None)]

notmasked_edges (a, axis=None)

Find the indices of the first and last unmasked values along an axis.

If all values are masked, return None. Otherwise, return a list of two tuples, corresponding to the indices of the first and last unmasked values respectively.

Parameters

a : array_like

The input array.

axis : int, optional

Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

Returns

dedges : ndarray or list

An array of start and end indexes if there are any masked data in the array. If there are no masked data in the array, edges is a list of the first and last index.

See Also:

flatnotmasked_contiguous, flatnotmasked_edges, notmasked_contiguous

Examples

>>> a = np.arange(9).reshape((3, 3))
>>> m = np.zeros_like(a)
>>> m[1:, 1:] = 1

1.6. Masked arrays
Modifying a mask

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numa.mask_cols(a[, axis])</code></td>
<td>Mask columns of a 2D array that contain masked values.</td>
</tr>
<tr>
<td><code>numa.mask_or(m1, m2[, copy, shrink])</code></td>
<td>Combine two masks with the <code>logical_or</code> operator.</td>
</tr>
<tr>
<td><code>numa.mask_rows(a[, axis])</code></td>
<td>Mask rows and/or columns of a 2D array that contain masked values.</td>
</tr>
<tr>
<td><code>numa.mask_rowcols(a[, axis])</code></td>
<td>Mask rows and/or columns of a 2D array.</td>
</tr>
<tr>
<td><code>numa.harden_mask(self)</code></td>
<td>Force the mask to hard.</td>
</tr>
<tr>
<td><code>numa.soften_mask(self)</code></td>
<td>Force the mask to soft.</td>
</tr>
<tr>
<td><code>numa.MaskedArray.harden_mask(self)</code></td>
<td>Reduce a mask to nomask when possible.</td>
</tr>
<tr>
<td><code>numa.MaskedArray.unshare_mask(self)</code></td>
<td>Copy the mask and set the sharedmask flag to False.</td>
</tr>
</tbody>
</table>

**`mask_cols(a[, axis=None])`**

Mask columns of a 2D array that contain masked values.

This function is a shortcut to `mask_rowcols` with `axis` equal to 1.

**See Also:**

- `mask_rowcols`  Mask rows and/or columns of a 2D array.
- `masked_where`  Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as numa
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = numa.masked_equal(a, 1)
>>> a
masked_array(data =
             [[0 0 0],
             [0 -- 0],
             [0 0 0]],
       mask =
             [[False False False],
             [False  True False],
             [False False False]],
      fill_value=999999)
>>> numa.mask_cols(a)
masked_array(data =
             [[0 -- 0],
             [0 -- 0],
```
[0 -- 0],
    mask =
[[False  True False]
 [False  True False]
 [False  True False]],
    fill_value=999999)

mask_or (m1, m2, copy=False, shrink=True)

Combine two masks with the logical_or operator.
The result may be a view on m1 or m2 if the other is nomask (i.e. False).

Parameters

m1, m2 : array_like
    Input masks.

copy : bool, optional
    If copy is False and one of the inputs is nomask, return a view of the other input mask.
    Defaults to False.

shrink : bool, optional
    Whether to shrink the output to nomask if all its values are False. Defaults to True.

Returns

mask : output mask
    The result masks values that are masked in either m1 or m2.

Raises

ValueError :
    If m1 and m2 have different flexible dtypes.

Examples

>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([ True,  True,  True, False], dtype=bool)

mask_rowcols (a, axis=None)

Mask rows and/or columns of a 2D array that contain masked values.

Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected
using the axis parameter.

• If axis is None, rows and columns are masked.
• If axis is 0, only rows are masked.
• If axis is 1 or -1, only columns are masked.

Parameters

a : array_like, MaskedArray
    The array to mask. If not a MaskedArray instance (or if no array elements are masked).
    The result is a MaskedArray with mask set to nomask (False). Must be a 2D array.

axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns

a : MaskedArray
A modified version of the input array, masked depending on the value of the axis parameter.

Raises

NotImplementedError :
If input array a is not 2D.

See Also:

mask_rows
Mask rows of a 2D array that contain masked values.

mask_cols
Mask cols of a 2D array that contain masked values.

masked_where
Mask where a condition is met.

Notes
The input array's mask is modified by this function.

Examples

>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
             [[0 0 0],
              [0 -- 0],
              [0 0 0]],
            mask =
             [[False False False],
              [False True False],
              [False False False]],
            fill_value=999999)
>>> ma.mask_rowcols(a)
masked_array(data =
             [[0 -- 0],
              [-- -- --],
              [0 -- 0]],
            mask =
             [[False True False],
              [True True True],
              [False True False]],
            fill_value=999999)
**mask_rows** *(a, axis=None)*

Mask rows of a 2D array that contain masked values.

This function is a shortcut to **mask_rowcols** with *axis* equal to 0.

See Also:

**mask_rowcols**

Mask rows and/or columns of a 2D array.

**masked_where**

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
    [[0 0 0],
     [0 -- 0],
     [0 0 0]],
  mask =
    [[False False False],
     [False  True False],
     [False False False]],
  fill_value=999999)
>>> ma.mask_rows(a)
masked_array(data =
    [[0 0 0],
     [-- -- --],
     [0 0 0]],
  mask =
    [[False False False],
     [ True  True  True],
     [False False False]],
  fill_value=999999)
```

**harden_mask** ()

Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its **hardmask** property. **harden_mask** sets **hardmask** to True.

See Also:

**hardmask**

**soften_mask** ()

Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its **hardmask** property. **soften_mask** sets **hardmask** to False.

See Also:
hardmask

harden_mask()  
Force the mask to hard.  

Whether the mask of a masked array is hard or soft is determined by its hardmask property.  
harden_mask sets hardmask to True.  

See Also:  
hardmask

soften_mask()  
Force the mask to soft.  

Whether the mask of a masked array is hard or soft is determined by its hardmask property.  
soften_mask sets hardmask to False.  

See Also:  
hardmask

shrink_mask()  
Reduce a mask to nomask when possible.

Parameters  
None :  

Returns  
None :  

Examples  

```python  
>>> x = np.ma.array([[1, 2], [3, 4]], mask=[0]*4)  
>>> x.mask  
array([[False, False],  
       [False, False]], dtype=bool)  
>>> x.shrink_mask()  
>>> x.mask  
False
```

unshare_mask()  
Copy the mask and set the sharedmask flag to False.  

Whether the mask is shared between masked arrays can be seen from the sharedmask property.  
unshare_mask ensures the mask is not shared.  
A copy of the mask is only made if it was shared.  

See Also:  
sharedmask
Conversion operations

> to a masked array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.asarray(a[, dtype, order])</code></td>
<td>Convert the input to a masked array of the given data-type.</td>
</tr>
<tr>
<td><code>ma.asanyarray(a[, dtype])</code></td>
<td>Convert the input to a masked array, conserving subclasses.</td>
</tr>
<tr>
<td><code>ma.fix_invalid(a[, mask, copy, fill_value])</code></td>
<td>Return input with invalid data masked and replaced by a fill value.</td>
</tr>
<tr>
<td><code>ma.masked_equal(x[, value[, copy]])</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater(x[, value[, copy]])</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater_equal(x[, value[, copy]])</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_inside(x, v1, v2[, copy])</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_invalid(a[, copy])</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>ma.masked_less(x[, value[, copy]])</code></td>
<td>Mask an array where less than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_less_equal(x[, value[, copy]])</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_not_equal(x[, value[, copy]])</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_object(x[, value[, copy, shrink]])</code></td>
<td>Mask the array x where the data are exactly equal to value.</td>
</tr>
<tr>
<td><code>ma.masked_outside(x, v1, v2[, copy])</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_values(x[, value[, rtol, atol, copy, ...]])</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>ma.masked_where(condition, a[, copy])</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
</tbody>
</table>

### asarray

Convert the input to a masked array of the given data-type.

No copy is performed if the input is already an `ndarray`. If `a` is a subclass of `MaskedArray`, a base class `MaskedArray` is returned.

**Parameters**

- **a**: array_like
  
  Input data, in any form that can be converted to a masked array. This includes lists, lists of tuples, tuples of tuples, tuples of lists, `ndarrays` and masked arrays.

- **dtype**: dtype, optional
  
  By default, the data-type is inferred from the input data.

- **order**: {'C', 'F'}, optional
  
  Whether to use row-major (‘C’) or column-major (‘FORTRAN’) memory representation. Default is ‘C’.

**Returns**

- **out**: MaskedArray
  
  Masked array interpretation of `a`.

**See Also**

- **asanyarray**
  
  Similar to `asarray`, but conserves subclasses.

**Examples**

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asarray(x)
masked_array(data =
```

1.6. Masked arrays
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asanyarray(x)
masked_array(data =
[][ 0. 1. 2. 3. 4.]
[ 5. 6. 7. 8. 9.],
mask =
False,
fill_value = 1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>

**asanyarray** (*a*, *dtype=None*)
Convert the input to a masked array, conserving subclasses.

If `a` is a subclass of `MaskedArray`, its class is conserved. No copy is performed if the input is already an *ndarray*.

**Parameters**
- **a**: array_like
  Input data, in any form that can be converted to an array.
- **dtype**: dtype, optional
  By default, the data-type is inferred from the input data.
- **order**: {'C', 'F'}, optional
  Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

**Returns**
- **out**: MaskedArray
  MaskedArray interpretation of `a`.

**See Also:**
- **asarray**
  Similar to `asanyarray`, but does not conserve subclass.

**Examples**

```python
given the array:
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])

>>> np.ma.asanyarray(x)  # return a MaskedArray
masked_array(data =
[][ 0. 1. 2. 3. 4.]
[ 5. 6. 7. 8. 9.],
mask =
False,
fill_value = 1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>
```

**fix_invalid** (*a*, *mask=False*, *copy=True*, *fill_value=None*)
Return input with invalid data masked and replaced by a fill value.

Invalid data means values of `nan`, `inf`, etc.

**Parameters**
- **a**: array_like
  Input array, a (subclass of) *ndarray*.
copy : bool, optional
Whether to use a copy of a (True) or to fix a in place (False). Default is True.

fill_value : scalar, optional
Value used for fixing invalid data. Default is None, in which case the a.fill_value is used.

Returns

b : MaskedArray
The input array with invalid entries fixed.

Notes
A copy is performed by default.

Examples

```python
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [-- -1.0 nan inf],
              mask = [ True False False False],
              fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data = [-- -1.0 -- --],
              mask = [ True False True True],
              fill_value = 1e+20)
```

```python
>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.00000000e+00, -1.00000000e+00, 1.00000000e+20, 1.00000000e+20])
```

```python
>>> x.data
array([ 1., -1., NaN, Inf])
```

masked_equal (x, value, copy=True)
Mask an array where equal to a given value.
This function is a shortcut to masked_where, with condition = (x == value). For floating point arrays, consider using masked_values(x, value).

See Also:

masked_where
Mask where a condition is met.

masked_values
Mask using floating point equality.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
              mask = [False False  True False])
```

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
```

1.6. Masked arrays
```python
mask = [False False True False],
fill_value=999999)
```

**masked_greater** *(x, value, copy=True)*

Mask an array where greater than a given value.

This function is a shortcut to `masked_where`, with `condition = (x > value)`.

See Also:

**masked_where**

Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data = [0 1 2 --],
mask = [False False False True],
fill_value=999999)
```

**masked_greater_equal** *(x, value, copy=True)*

Mask an array where greater than or equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x >= value)`.

See Also:

**masked_where**

Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 -- --],
mask = [False False True True],
fill_value=999999)
```

**masked_inside** *(x, v1, v2, copy=True)*

Mask an array inside a given interval.

Shortcut to `masked_where`, where `condition` is True for `x` inside the interval [v1,v2] (v1 <= x <= v2). The boundaries v1 and v2 can be given in either order.

See Also:

**masked_where**

Mask where a condition is met.
Notes

The array $x$ is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
            fill_value=1e+20)
```

The order of $v1$ and $v2$ doesn’t matter.

```python
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
            fill_value=1e+20)
```

```python
masked_invalid(a, copy=True)
```

Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to `masked_where`, with `condition = ~(np.isfinite(a))`. Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

See Also:

```python
masked_where
```

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0., 1., NaN, Inf, 4.])
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
             mask = [False False True True False],
            fill_value=1e+20)
```

```python
masked_less(x, value, copy=True)
```

Mask an array where less than a given value.

This function is a shortcut to `masked_where`, with `condition = (x < value)`.

See Also:

```python
masked_where
```

Mask where a condition is met.
Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
masked_array(data = [-- -- 2 3],
          mask = [ True  True False False],
          fill_value=999999)

**mask<inf>ed_<inf>_<inf>_<inf>_less<inf>_equal<inf>** (x, value, copy=True)
Mask an array where less than or equal to a given value.

This function is a shortcut to masked<inf>where<inf>, with condition = (x <= value).

See Also:

**masked<inf>_where**
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
masked_array(data = [-- -- -- 3],
             mask = [ True  True  True False],
             fill_value=999999)

**mask<inf>ed_<inf>_<inf>_<inf>_not<inf>_equal<inf>** (x, value, copy=True)
Mask an array where not equal to a given value.

This function is a shortcut to masked<inf>where<inf>, with condition = (x != value).

See Also:

**masked<inf>_where**
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data = [-- -- 2 --],
             mask = [ True  True False  True],
             fill_value=999999)

**mask<inf>ed_<inf>_<inf>_<inf>_object** (x, value, copy=True, shrink=True)
Mask the array x where the data are exactly equal to value.

This function is similar to masked<inf>values<inf>, but only suitable for object arrays: for floating point, use masked<inf>values<inf> instead.
Parameters

- **x**: array_like
  - Array to mask

- **value**: object
  - Comparison value

- **copy**: {True, False}, optional
  - Whether to return a copy of `x`.

- **shrink**: {True, False}, optional
  - Whether to collapse a mask full of False to nomask

Returns

- **result**: MaskedArray
  - The result of masking `x` where equal to `value`.

See Also:

- **masked_where**
  - Mask where a condition is met.

- **masked_equal**
  - Mask where equal to a given value (integers).

- **masked_values**
  - Mask using floating point equality.

Examples

```python
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> print eat
[-- ham]
>>> # plain ol' ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> print eat
[cheese ham pineapple]
```

Note that `mask` is set to `nomask` if possible.

```python
>>> eat
masked_array(data = [cheese ham pineapple],
    mask = False,
    fill_value=?)
```

**masked_outside** *(x, v1, v2, copy=True)*

Mask an array outside a given interval.

Shortcut to `masked_where`, where `condition` is True for `x` outside the interval [v1,v2] (x < v1)(x > v2). The boundaries v1 and v2 can be given in either order.

See Also:
masked_where

Mask where a condition is met.

Notes

The array $x$ is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
              mask = [ True True False False True True],
              fill_value=1e+20)
```

The order of $v1$ and $v2$ doesn’t matter.

```python
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
              mask = [ True True False False True True],
              fill_value=1e+20)
```

masked_values

Mask using floating point equality.

Return a MaskedArray, masked where the data in array $x$ are approximately equal to $value$, i.e. where the following condition is True

$$(\text{abs}(x - value) <= \text{atol}+\text{rtol} \cdot \text{abs(value)})$$

The fill_value is set to $value$ and the mask is set to nomask if possible. For integers, consider using masked_equal.

Parameters

- $x$: array_like
  Array to mask.
- $value$: float
  Masking value.
- $rtol$: float, optional
  Tolerance parameter.
- $atol$: float, optional
  Tolerance parameter (1e-8).
- $copy$: bool, optional
  Whether to return a copy of $x$.
- $shrink$: bool, optional
  Whether to collapse a mask full of False to nomask.

Returns

- $result$: MaskedArray
  The result of masking $x$ where approximately equal to $value$. 

202 Chapter 1. Array objects
**See Also:**

masked_where  
Mask where a condition is met.

masked_equal  
Mask where equal to a given value (integers).

**Examples**

```python
>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
mask = [False True False True False],
fill_value=1.1)
```

Note that `mask` is set to `nomask` if possible.

```python
>>> ma.masked_values(x, 1.5)
masked_array(data = [ 1. 1.1 2. 1.1 3. ],
mask = False,
fill_value=1.5)
```

For integers, the fill value will be different in general to the result of `masked_equal`.

```python
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data = [0 1 -- 3 4],
mask = [False False True False False],
fill_value=2)
>>> ma.masked_equal(x, 2)
masked_array(data = [0 1 -- 3 4],
mask = [False False True False False],
fill_value=999999)
```

**masked_where** *(condition, a, copy=True)*  
Mask an array where a condition is met.

Return `a` as an array masked where `condition` is True. Any masked values of `a` or `condition` are also masked in the output.

**Parameters**

- **condition** : array_like  
  Masking condition. When `condition` tests floating point values for equality, consider using `masked_values` instead.

- **a** : array_like  
  Array to mask.

- **copy** : bool  
  If True (default) make a copy of `a` in the result. If False modify `a` in place and return a view.
Returns

result : MaskedArray

The result of masking a where condition is True.

See Also:

masked_values
Mask using floating point equality.

masked_equal
Mask where equal to a given value.

masked_not_equal
Mask where not equal to a given value.

masked_less_equal
Mask where less than or equal to a given value.

masked_greater_equal
Mask where greater than or equal to a given value.

masked_less
Mask where less than a given value.

masked_greater
Mask where greater than a given value.

masked_inside
Mask inside a given interval.

masked_outside
Mask outside a given interval.

masked_invalid
Mask invalid values (NaNs or infs).

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
               mask = [ True True True False],
               fill_value=999999)
```

Mask array b conditional on a.

```python
>>> b = ['a', 'b', 'c', 'd']
>>> ma.masked_where(a == 2, b)
masked_array(data = ['a' 'b' -- 'd'],
               mask = [False False True False],
               fill_value=N/A)
```

Effect of the copy argument.
>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data = [-- -- -- 3],
             mask = [ True True True False],
             fill_value=999999)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
             mask = [False True True False],
             fill_value=999999)

When condition or a contain masked values.

>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data = [0 1 -- 3],
             mask = [False False True False],
             fill_value=999999)

> to a ndarray

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.compress_cols(a)</td>
<td>Suppress whole columns of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td>ma.compress_rowcols(x[, axis])</td>
<td>Suppress the rows and/or columns of a 2-D array that contain</td>
</tr>
<tr>
<td></td>
<td>masked values.</td>
</tr>
<tr>
<td></td>
<td>Suppress whole rows of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td></td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td>ma.compressed(x)</td>
<td>Return input as an array with masked data replaced by a fill value.</td>
</tr>
<tr>
<td>ma.filled(a[, fill_value])</td>
<td>Return a copy of self, where masked values are filled with a fill value.</td>
</tr>
</tbody>
</table>

**compress_cols(a)**

Suppress whole columns of a 2-D array that contain masked values.

This is equivalent to np.maextras.compress_rowcols(a, 1), see extras.compress_rowcols for details.
compress_rowcols

Suppress the rows and/or columns of a 2-D array that contain masked values.

The suppression behavior is selected with the `axis` parameter.

- If `axis` is None, both rows and columns are suppressed.
- If `axis` is 0, only rows are suppressed.
- If `axis` is 1 or -1, only columns are suppressed.

**Parameters**

- `axis` : int, optional
  
  Axis along which to perform the operation. Default is None.

**Returns**

- `compressed_array` : ndarray
  
  The compressed array.

**Examples**

```python
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
... [1, 0, 0],
... [0, 0, 0]])
>>> x
masked_array(data =
  [[-- 1 2]
  [-- 4 5]
  [6 7 8]],
  mask =
  [[ True False False]
  [ True False False]
  [False False False]],
  fill_value = 999999)
```

```python
>>> np.ma.extras.compress_rowcols(x)
array([[7, 8]])
```

```python
>>> np.ma.extras.compress_rowcols(x, 0)
array([[6, 7, 8]])
```

```python
>>> np.ma.extras.compress_rowcols(x, 1)
array([[1, 2],
  [4, 5],
  [7, 8]])
```

**compress_rows**

Suppress whole rows of a 2-D array that contain masked values.

This is equivalent to `np.ma.extras.compress_rowcols(a, 0)`, see `extras.compress_rowcols` for details.

**See Also:**

- `extras.compress_rowcols`
**compressed**(*x*)

Return all the non-masked data as a 1-D array.

This function is equivalent to calling the "compressed" method of a *MaskedArray*, see *MaskedArray.compressed* for details.

**See Also:**

*MaskedArray.compressed*

Equivalent method.

**filled**(a, *fill_value=None*)

Return input as an array with masked data replaced by a fill value.

If *a* is not a *MaskedArray*, *a* itself is returned. If *a* is a *MaskedArray* and *fill_value* is None, *fill_value* is set to *a.fill_value*.

**Parameters**

- **a**: MaskedArray or array_like
  
  An input object.

- **fill_value**: scalar, optional
  
  Filling value. Default is None.

**Returns**

- **a**: ndarray
  
  The filled array.

**Examples**

```python
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
... [1, 0, 0],
... [0, 0, 0]])
>>> x.filled()
array([[999999, 1, 2],
       [999999, 4, 5],
       [ 6, 7, 8]])
```

**compressed**()

Return all the non-masked data as a 1-D array.

**Returns**

- **data**: ndarray
  
  A new *ndarray* holding the non-masked data is returned.

**Notes**

The result is not a *MaskedArray*!

**Examples**

```python
>>> x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
>>> x.compressed()
array([0, 1])
>>> type(x.compressed())
<type 'numpy.ndarray'>
```
filled\( (\text{fill\_value=n}0\text{ne}) \)  
Return a copy of self, where masked values are filled with a fill value.  

If fill_value is None, MaskedArray.fill_value is used instead.  

Parameters  
fill_value : scalar, optional  

The value to use for invalid entries. Default is None.  

Notes  
The result is not a MaskedArray!  

Examples  
```python  
>>> x = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 0, 1], fill_value=-999)  
>>> x.filled()  
array([1, 2, -999, 4, -999])  
>>> type(x.filled())  
<type 'numpy.ndarray'>  
```

Subclassing is preserved. This means that if the data part of the masked array is a matrix, filled returns a matrix:  
```python  
>>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])  
>>> x.filled()  
matrix([[ 1, 999999],  
[999999, 4]])  
```

> to another object  
| ma.MaskedArray.tofile (self, fid[, sep, format]) | Save a masked array to a file in binary format. |
| ma.MaskedArray.tolist (self[, fill_value]) | Return the data portion of the masked array as a hierarchical Python list. Transforms a masked array into a flexible-type array. |
| ma.MaskedArray.torecords (self) | Return the array data as a string containing the raw bytes in the array. |

| tofile (fid, sep=",", format='%s') | Save a masked array to a file in binary format. |

Warning: This function is not implemented yet.  

Raises  
NotImplementedError :  
When tofile is called.  

tolist\( (\text{fill\_value=n}0\text{ne}) \)  
Return the data portion of the masked array as a hierarchical Python list.  

Data items are converted to the nearest compatible Python type. Masked values are converted to fill_value. If fill_value is None, the corresponding entries in the output list will be None.  

Parameters  
fill_value : scalar, optional  

The value to use for invalid entries. Default is None.
Returns
result : list

The Python list representation of the masked array.

Examples

```python
given_array = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0, 1, 0] * 4)
given_array.tolist()
given_array.tolist(-999)
```

```python
torecords()

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

- the _data field stores the _data part of the array.
- the _mask field stores the _mask part of the array.

Parameters

None :

Returns

record : ndarray

A new flexible-type ndarray with two fields: the first element containing a value, the
second element containing the corresponding mask boolean. The returned record shape
matches self.shape.

Notes

A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value,
...) will be lost.

Examples

```python
given_array = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0, 1, 0] * 4)
given_array.toflex()
```

```python
tostring(fill_value=None, order='C')

Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

Parameters

fill_value : scalar, optional

Value used to fill in the masked values. Default is None, in which case MaskedArray.fill_value is used.

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

1.6. Masked arrays
Order of the data item in the copy. Default is ‘C’.

- ‘C’ – C order (row major).
- ‘F’ – Fortran order (column major).
- ‘A’ – Any, current order of array.
- None – Same as ‘A’.

See Also:

- `ndarray.tostring`, `tolist`, `tofile`

Notes

As for `ndarray.tostring`, information about the shape, dtype, etc., but also about fill_value, will be lost.

Examples

```python
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tostring()
'\x01\x00\x00\x00?B\x0f\x00?B\x0f\x00\x04\x00\x04\x00\x00'
```

Pickling and unpickling

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.dump</code></td>
<td>Pickle a masked array to a file.</td>
</tr>
<tr>
<td><code>ma.dumps</code></td>
<td>Return a string corresponding to the pickling of a masked array.</td>
</tr>
<tr>
<td><code>ma.load</code></td>
<td>Wrapper around <code>cPickle.load</code> which accepts either a file-like object or a filename.</td>
</tr>
<tr>
<td><code>ma.loads</code></td>
<td>Load a pickle from the current string.</td>
</tr>
</tbody>
</table>

`dump(a, F)`

Pickle a masked array to a file.

This is a wrapper around `cPickle.dump`.

Parameters

- `a` : MaskedArray
  - The array to be pickled.
- `F` : str or file-like object
  - The file to pickle `a` to. If a string, the full path to the file.

`dumps(a)`

Return a string corresponding to the pickling of a masked array.

This is a wrapper around `cPickle.dumps`.

Parameters

- `a` : MaskedArray
  - The array for which the string representation of the pickle is returned.

`load(F)`

Wrapper around `cPickle.load` which accepts either a file-like object or a filename.

Parameters

- `F` : str or file
  - The file or file name to load.
dump

Pickle an array

Notes

This is different from `numpy.load`, which does not use cPickle but loads the NumPy binary `.npy` format.

loads (`strg`)

Load a pickle from the current string.

The result of `cPickle.loads(strg)` is returned.

Parameters

- `strg` : str
  The string to load.

See Also:

dumps

Return a string corresponding to the pickling of a masked array.

Filling a masked array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.common_fill_value(a, b)</code></td>
<td>Return the common filling value of two masked arrays, if any.</td>
</tr>
<tr>
<td><code>ma.default_fill_value(obj)</code></td>
<td>Return the default fill value for the argument object.</td>
</tr>
<tr>
<td><code>ma.maximum_fill_value(obj)</code></td>
<td>Return the minimum value that can be represented by the dtype of an object.</td>
</tr>
<tr>
<td><code>ma.maximum_fill_value(obj)</code></td>
<td>Return the minimum value that can be represented by the dtype of an object.</td>
</tr>
<tr>
<td><code>ma.set_fill_value(a, fill_value)</code></td>
<td>Set the filling value of a, if a is a masked array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.get_fill_value(self)</code></td>
<td>Return the filling value of the masked array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.fill_value</code></td>
<td>Filling value.</td>
</tr>
</tbody>
</table>

common_fill_value (`a`, `b`)

Return the common filling value of two masked arrays, if any.

If `a.fill_value == b.fill_value`, return the fill value, otherwise return `None`.

Parameters

- `a`, `b` : MaskedArray
  The masked arrays for which to compare fill values.

Returns

- `fill_value` : scalar or `None`
  The common fill value, or `None`.

Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=3)
>>> y = np.ma.array([0, 1.], fill_value=3)
>>> np.ma.common_fill_value(x, y)
3.0
```

default_fill_value (`obj`)

Return the default fill value for the argument object.

The default filling value depends on the datatype of the input array or the type of the input scalar:
<table>
<thead>
<tr>
<th>datatype</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>int</td>
<td>999999</td>
</tr>
<tr>
<td>float</td>
<td>1e20</td>
</tr>
<tr>
<td>complex</td>
<td>1e20+0j</td>
</tr>
<tr>
<td>object</td>
<td>‘?’</td>
</tr>
<tr>
<td>string</td>
<td>‘N/A’</td>
</tr>
</tbody>
</table>

Parameters

- **obj**: ndarray, dtype or scalar
  
The array data-type or scalar for which the default fill value is returned.

Returns

- **fill_value**: scalar
  
The default fill value.

Examples

```python
>>> np.ma.default_fill_value(1)
999999
>>> np.ma.default_fill_value(np.array([1.1, 2., np.pi]))
1e+20
>>> np.ma.default_fill_value(np.dtype(complex))
(1e+20+0j)
```

maximum_fill_value(obj)

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

Parameters

- **obj**: ndarray, dtype
  
An object that can be queried for it’s numeric type.

Returns

- **val**: scalar
  
The minimum representable value.

Raises

- **TypeError**
  
If *obj* isn’t a suitable numeric type.

See Also:

- **minimum_fill_value**
  
The inverse function.

- **set_fill_value**
  
Set the filling value of a masked array.

- **MaskedArray.fill_value**
  
Return current fill value.
Examples

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
```

An array of numeric data can also be passed.

```python
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf
```

**maximum_fill_value**(obj)

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

**Parameters**

- **obj**: {ndarray, dtype}
  
  An object that can be queried for it’s numeric type.

**Returns**

- **val**: scalar
  
  The minimum representable value.

**Raises**

- **TypeError**:
  
  If `obj` isn’t a suitable numeric type.

**See Also**

- **minimum_fill_value**
  
  The inverse function.

- **set_fill_value**
  
  Set the filling value of a masked array.

- **MaskedArray.fill_value**
  
  Return current fill value.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
```

An array of numeric data can also be passed.
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf

**set_fill_value**(*a*, *fill_value*)

Set the filling value of a, if a is a masked array.

This function changes the fill value of the masked array *a* in place. If *a* is not a masked array, the function returns silently, without doing anything.

**Parameters**

- **a**: array_like
  Input array.

- **fill_value**: dtype
  Filling value. A consistency test is performed to make sure the value is compatible with the dtype of *a*.

**Returns**

- None
  Nothing returned by this function.

**See Also:**

- **maximum_fill_value**
  Return the default fill value for a dtype.

- **MaskedArray.fill_value**
  Return current fill value.

- **MaskedArray.set_fill_value**
  Equivalent method.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a = ma.masked_where(a < 3, a)
>>> a
masked_array(data = [-- -- -- 3 4],
              mask = [ True True True False False],
              fill_value=999999)
>>> ma.set_fill_value(a, -999)
>>> a
masked_array(data = [-- -- -- 3 4],
              mask = [ True True True False False],
              fill_value=-999)
```

Nothing happens if *a* is not a masked array.
```python
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> ma.set_fill_value(a, 100)
>>> a
[0, 1, 2, 3, 4]
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> ma.set_fill_value(a, 100)
>>> a
array([0, 1, 2, 3, 4])
```

### get_fill_value()

Return the filling value of the masked array.

**Returns**

- **fill_value**: scalar
  
The filling value.

### Examples

```python
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...   np.ma.array([0, 1], dtype=dt).get_fill_value()
...
999999
999999
1e+20
(1e+20+0j)
```

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.get_fill_value()
-inf
```

### set_fill_value(value=None)

Set the filling value of the masked array.

**Parameters**

- **value**: scalar, optional
  
The new filling value. Default is None, in which case a default based on the data type is used.

**See Also:**

- **ma.set_fill_value**: Equivalent function.

### Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.set_fill_value(np.pi)
>>> x.fill_value
3.1415926535897931
```
Reset to default:

```python
>>> x.set_fill_value()
>>> x.fill_value
1e+20
```

**fill_value**

Filling value.

## Masked arrays arithmetics

### Arithmetics

```
ma.anom (self, axis = None, dtype = None)  
Compute the anomalies (deviations from the arithmetic mean) along the given axis. 

ma.anomalies (self, axis = None, keepdims = False)  
Compute the anomalies (deviations from the arithmetic mean) along the given axis. 

ma.average (a, axis = None, weights = None, returned = False)  
Return the weighted average of array over the given axis. 

ma.conjugate (x, out = None)  
Return the complex conjugate, element-wise. 

ma.corrcoef (x, y = None, rowvar = True, bias = False, ddof = None, axis = 0)  
Return correlation coefficients of the input array. 

ma.cov (x, y = None, rowvar = True, bias = False, ddof = None, fweights = None, aweights = None, bweights = None, axis = 0)  
Estimate the covariance matrix. 

ma.cumsum (self, axis = 0, dtype = None, out = None)  
Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis. 

ma.cumprod (self, axis = 0, dtype = None, out = None)  
Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis. 

ma.mean (a, axis = None, dtype = None, out = None, keepdims = False)  
Return the average of the array elements. 

ma.median (a, axis = None, out = None, overwrite_input = False)  
Compute the median along the specified axis. 

ma.power (a, b, third = False)  
Returns element-wise base array raised to power from second array. 

ma.prod (self, axis = None, dtype = None, out = None, keepdims = False)  
Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation. 

ma.std (a, axis = None, dtype = None, out = None, ddof = None, keepdims = False, mask = None, overwrite_input = False)  
Compute the standard deviation along the specified axis. 

ma.sum (self, axis = None, dtype = None, out = None, keepdims = False, initial = None)  
Return the sum of the array elements over the given axis. Masked elements are set to 0 internally. 

ma.var (a, axis = None, dtype = None, out = None, ddof = None, keepdims = False, mask = None, overwrite_input = False)  
Compute the variance along the specified axis. 

ma.MaskedArray.anom (self, axis = None, dtype = None)  
Compute the anomalies (deviations from the arithmetic mean) along the given axis. 

ma.MaskedArray.anomalies (self, axis = None, keepdims = False)  
Compute the anomalies (deviations from the arithmetic mean) along the given axis. 

ma.MaskedArray.cumsum (self, axis = 0, dtype = None, out = None)  
Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis. 

ma.MaskedArray.cumprod (self, axis = 0, dtype = None, out = None)  
Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis. 

ma.MaskedArray.mean (self, axis = None, dtype = None, out = None, overwrite_input = False)  
Returns the average of the array elements. 

ma.MaskedArray.prod (self, axis = None, dtype = None, out = None, overwrite_input = False)  
Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation. 

ma.MaskedArray.std (self, axis = None, dtype = None, out = None, ddof = None, keepdims = False, overwrite_input = False, mask = None)  
Compute the standard deviation along the specified axis. 

ma.MaskedArray.sum (self, axis = None, dtype = None, out = None, overwrite_input = False)  
Return the sum of the array elements over the given axis. Masked elements are set to 0 internally. 

ma.MaskedArray.var (self, axis = None, dtype = None, out = None, ddof = None, overwrite_input = False, mask = None)  
Compute the variance along the specified axis.
```
anom()  
Compute the anomalies (deviations from the arithmetic mean) along the given axis.
Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters
axis : int, optional
    Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional
    Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:
mean
    Compute the mean of the array.

Examples

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
    mask = False,
    fill_value = 1e+20)
```

anomalies()  
Compute the anomalies (deviations from the arithmetic mean) along the given axis.
Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters
axis : int, optional
    Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional
    Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:
mean
    Compute the mean of the array.

Examples

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
    mask = False,
    fill_value = 1e+20)
```
average(a, axis=None, weights=None, returned=False)

Return the weighted average of array over the given axis.

Parameters
   a : array_like
       Data to be averaged. Masked entries are not taken into account in the computation.

   axis : int, optional
       Axis along which the variance is computed. The default is to compute the variance of the flattened array.

   weights : array_like, optional
       The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of a along the given axis) or of the same shape as a. If weights=None, then all data in a are assumed to have a weight equal to one.

   returned : bool, optional
       Flag indicating whether a tuple (result, sum of weights) should be returned as output (True), or just the result (False). Default is False.

Returns
   average, [sum_of_weights] : (tuple of) scalar or MaskedArray
       The average along the specified axis. When returned is True, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is np.float64 if a is of integer type, otherwise it is of the same type as a. If returned, sum_of_weights is of the same type as average.

Examples

>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25

>>> x = np.ma.arange(6.).reshape(3, 2)
>>> print x
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]]
>>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3],
... returned=True)
>>> print avg
[2.66666666667 3.66666666667]

conjugate()

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters
   x : array_like
       Input value.

Returns
   y : ndarray
The complex conjugate of $x$, with same dtype as $y$.

**Examples**

```python
>>> np.conjugate(1+2j)
(1-2j)
```

```python
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j, 0.-0.j],
       [ 0.-0.j, 1.-1.j]])
```

corrcoef $(x, y=None, rowvar=True, bias=False, allow_masked=True)$

Return correlation coefficients of the input array.

Except for the handling of missing data this function does the same as `numpy.corrcoef`. For more details and examples, see `numpy.corrcoef`.

**Parameters**

- **x**: array_like
  A 1-D or 2-D array containing multiple variables and observations. Each row of $x$ represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.

- **y**: array_like, optional
  An additional set of variables and observations. $y$ has the same shape as $x$.

- **rowvar**: bool, optional
  If `rowvar` is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

- **bias**: bool, optional
  Default normalization (False) is by $(N-1)$, where $N$ is the number of observations given (unbiased estimate). If `bias` is 1, then normalization is by $N$.

- **allow_masked**: bool, optional
  If True, masked values are propagated pair-wise: if a value is masked in $x$, the corresponding value is masked in $y$. If False, raises an exception.

**See Also:**

- `numpy.corrcoef`
  Equivalent function in top-level NumPy module.

- `cov`
  Estimate the covariance matrix.

cov $(x, y=None, rowvar=True, bias=False, allow_masked=True)$

Estimate the covariance matrix.

Except for the handling of missing data this function does the same as `numpy.cov`. For more details and examples, see `numpy.cov`.

By default, masked values are recognized as such. If $x$ and $y$ have the same shape, a common mask is allocated: if $x[i, j]$ is masked, then $y[i, j]$ will also be masked. Setting `allow_masked` to False will raise an exception if values are missing in either of the input arrays.
### Parameters

- **x**: array_like
  A 1-D or 2-D array containing multiple variables and observations. Each row of $x$ represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.

- **y**: array_like, optional
  An additional set of variables and observations. $y$ has the same form as $x$.

- **rowvar**: bool, optional
  If `rowvar` is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

- **bias**: bool, optional
  Default normalization (False) is by $(N-1)$, where $N$ is the number of observations given (unbiased estimate). If `bias` is True, then normalization is by $N$.

- **allow_masked**: bool, optional
  If True, masked values are propagated pair-wise: if a value is masked in $x$, the corresponding value is masked in $y$. If False, raises a `ValueError` exception when some values are missing.

### Raises

- **ValueError**: 
  Raised if some values are missing and `allow_masked` is False.

### See Also:

- `numpy.cov`

#### `cumsum()`

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

### Parameters

- **axis**: {None, -1, int}, optional
  Axis along which the sum is computed. The default (`axis = None`) is to compute over the flattened array. `axis` may be negative, in which case it counts from the last to the first axis.

- **dtype**: {None, dtype}, optional
  Type of the returned array and of the accumulator in which the elements are summed. If `dtype` is not specified, it defaults to the dtype of $a$, unless $a$ has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

### Returns

- **cumsum**: ndarray.
A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

Notes

The mask is lost if `out` is not a valid `MaskedArray`!

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

`cumprod()`

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

- `axis` : {None, -1, int}, optional
  
  Axis along which the product is computed. The default (`axis = None`) is to compute over the flattened array.

- `dtype` : {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the `dtype` is the same as that of `a`.

- `out` : ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

- `cumprod` : ndarray
  
  A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

Notes

The mask is lost if `out` is not a valid `MaskedArray`!

Arithmetic is modular when using integer types, and no error is raised on overflow.

`mean()`

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to `numpy.mean` for the full documentation.

Parameters

- `a` : array_like

  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.
axis : int, optional
Axis along which the means are computed. The default is to compute the mean of the flattened array.
dtype : dtype, optional
Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.
out : ndarray, optional
Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns
mean : ndarray, see dtype parameter above
If out=None, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also:

numpy.ma.mean
Equivalent function.
numpy.mean
Equivalent function on non-masked arrays.
numpy.ma.average
Weighted average.

Examples

```python
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
             mask = [False False True],
            fill_value = 999999)
>>> a.mean()
1.5
```

median(a, axis=None, out=None, overwrite_input=False)
Compute the median along the specified axis.

Returns the median of the array elements.

Parameters

- a : array_like
  Input array or object that can be converted to an array.
- axis : int, optional
  Axis along which the medians are computed. The default (None) is to compute the median along a flattened version of the array.
- out : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.
- overwrite_input : bool, optional
If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is True, and the input is not already an ndarray, an error will be raised.

Returns

median : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return data-type is float64 for integers and floats smaller than float64, or the input data-type, otherwise.

See Also:

mean

Notes

Given a vector V with N non masked values, the median of V is the middle value of a sorted copy of V (Vs) - i.e. Vs[(N-1)/2], when N is odd, or {Vs[N/2 - 1] + Vs[N/2]}/2 when N is even.

Examples

```python
>>> x = np.ma.array(np.arange(8), mask=[0]*4 + [1]*4)
>>> np.ma.extras.median(x)
1.5

>>> x = np.ma.array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)
>>> np.ma.extras.median(x)
2.5
>>> np.ma.extras.median(x, axis=-1, overwrite_input=True)
masked_array(data = [ 2.  5.],
           mask = False,
           fill_value = 1e+20)
```

power (a, b, third=None)

Returns element-wise base array raised to power from second array.

This is the masked array version of numpy.power. For details see numpy.power.

See Also:

numpy.power

Notes

The out argument to numpy.power is not supported, third has to be None.

prod ()

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

axis : {None, int}, optional

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of `a`.

**out** : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

`product_along_axis` : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as `a` with the specified axis removed. Returns a 0d array when `a` is 1d or `axis=None`. Returns a reference to the specified output array if specified.

**See Also:**

`prod`

Equivalent function

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> np.prod([1., 2.])
2.0
>>> np.prod([1., 2.], dtype=np.int32)
2
>>> np.prod([[1., 2.], [3., 4.]])
24.0
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([ 2., 12.])
```

**std()**

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

- `a` : array_like
  
  Calculate the standard deviation of these values.

- `axis` : int, optional
  
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- `dtype` : dtype, optional
  
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- `out` : ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.
ddof : int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is $N - ddof$, where $N$ represents the number of elements. By default $ddof$ is zero.

Returns

standard_deviation : ndarray, see dtype parameter above.

If out is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

var, mean

numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., $\text{std} = \sqrt{\text{mean}(|x - \text{mean}(x)|^2)}$.

The mean is normally calculated as $x.\text{sum()} / N$, where $N = \text{len}(x)$. If, however, $ddof$ is specified, the divisor $N - ddof$ is used instead. In standard statistical practice, $ddof=1$ provides an unbiased estimator of the variance of the infinite population. $ddof=0$ provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with $ddof=1$, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, $\text{std}$ takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the $\text{std}$ is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the $\text{dtype}$ keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1., 1.])
>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

In single precision, $\text{std}()$ can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0, :]
array(1.0)
>>> a[1, :]
array(0.1)
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

sum()

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
Parameters

axis : {None, -1, int}, optional

Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, ndarray}, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

sum_along_axis : MaskedArray or scalar

An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

Examples

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
```

```python
>>> print x.sum()
25
```

```python
>>> print x.sum(axis=1)
[4 5 16]
```

```python
>>> print x.sum(axis=0)
[8 5 12]
```

```python
>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type 'numpy.int64'>
```

var()

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

a : array_like

Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

axis : int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : dtype, optional

Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.


**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type is cast if necessary.

**ddof**: int, optional

“Delta Degrees of Freedom”: the divisor used in calculation is \( N - ddof \), where \( N \) represents the number of elements. By default \( ddof \) is zero.

**Returns**

**variance**: ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise a reference to the output array is returned.

**See Also**:

**std**

Standard deviation

**mean**

Average

**numpy.doc.ufuncs**

Section “Output arguments”

**Notes**

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x\text{.mean()}^2) \).

The mean is normally calculated as \( x\text{.sum()} / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of the infinite population. \( ddof=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( dtype \) keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, 0)
array([ 1.,  1.])
>>> np.var(a, 1)
array([ 0.25,  0.25])
```

In single precision, \( \text{var}() \) can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.var(a)
0.20405951142311096
```
Computing the standard deviation in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

`anom(axis=None, dtype=None)`
Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**

- `axis` : int, optional
  Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

- `dtype` : dtype, optional
  Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

**See Also:**

`mean`  
Compute the mean of the array.

**Examples**

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
             mask = False,
            fill_value = 1e+20)
```

`cumprod(axis=None, dtype=None, out=None)`
Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

- `axis` : {None, -1, int}, optional
  Axis along which the product is computed. The default (axis = None) is to compute over the flattened array.

- `dtype` : {None, dtype}, optional
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- `out` : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.
Returns
cumprod : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes
The mask is lost if out is not a valid MaskedArray!
Arithmetic is modular when using integer types, and no error is raised on overflow.

cumsum (axis=None, dtype=None, out=None)
Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters
axis : {None, -1, int}, optional
Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array. axis may be negative, in which case it counts from the last to the first axis.
dtype : {None, dtype}, optional
Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.
out : ndarray, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns
cumsum : ndarray.
A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes
The mask is lost if out is not a valid MaskedArray!
Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

mean (axis=None, dtype=None, out=None)
Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to numpy.mean for the full documentation.

1.6. Masked arrays
Parameters

- **a**: array_like
  Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

- **axis**: int, optional
  Axis along which the means are computed. The default is to compute the mean of the flattened array.

- **dtype**: dtype, optional
  Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

- **mean**: ndarray, see dtype parameter above
  If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also:

- `numpy.ma.mean`
  Equivalent function.

- `numpy.mean`
  Equivalent function on non-masked arrays.

- `numpy.ma.average`
  Weighted average.

Examples

```python
def a
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> a.mean()
1.5
```

**prod** (*axis=None, dtype=None, out=None*)

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

- **axis**: {None, int}, optional
  Axis over which the product is taken. If None is used, then the product is over all the array elements.

- **dtype**: {None, dtype}, optional
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If *dtype* has the value *None* and the type of *a* is an integer type of precision
less than the default platform integer, then the default platform integer precision is used.
Otherwise, the dtype is the same as that of a.

out : {None, array}, optional
Alternative output array in which to place the result. It must have the same shape as the
expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns
a 0d array when a is 1d or axis=None. Returns a reference to the specified output array
if specified.

See Also:

prod
equivalent function

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> np.prod([1.,2.])
2.0
>>> np.prod([1.,2.], dtype=np.int32)
2
>>> np.prod([[1.,2.],[3.,4.]])
24.0
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])
```

std (axis=None, dtype=None, out=None, ddof=0)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard
deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

a : array_like
Calculate the standard deviation of these values.

axis : int, optional
Axis along which the standard deviation is computed. The default is to compute the
standard deviation of the flattened array.

dtype : dtype, optional
Type to use in computing the standard deviation. For arrays of integer type the default
is float64, for arrays of float types it is the same as the array type.

out : ndarray, optional
Alternative output array in which to place the result. It must have the same shape as the
expected output but the type (of the calculated values) will be cast if necessary.

ddof : int, optional
Means Delta Degrees of Freedom. The divisor used in calculations is $N - ddof$, where $N$ represents the number of elements. By default $ddof$ is zero.

Returns

standard_deviation : ndarray, see dtype parameter above.

If $out$ is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

var, mean

numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., $std = \sqrt{\text{mean}(\text{abs}(x - \text{x.mean()})^2)}$.

The mean is normally calculated as $x . \text{sum()} / N$, where $N = \text{len}(x)$. If, however, $ddof$ is specified, the divisor $N - ddof$ is used instead. In standard statistical practice, $ddof=1$ provides an unbiased estimator of the variance of the infinite population. $ddof=0$ provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with $ddof=1$, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, $std$ takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the $std$ is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the $dtype$ keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([1., 1.])
>>> np.std(a, axis=1)
array([0.5, 0.5])
```

In single precision, std() can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

sum (axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
Parameters

- **axis**: {None, -1, int}, optional
  
  Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out**: {None, ndarray}, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

- **sum_along_axis**: MaskedArray or scalar
  
  An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

Examples

```python
def main():
    a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=np.float64)
    print('a:', a)
    print('a.sum():', a.sum())
    print('a.sum(axis=1):', a.sum(axis=1))
    print('a.sum(axis=0):', a.sum(axis=0))
```

```bash
$ python main.py
a: [[ 1.  2.  3.] [ 4.  5.  6.] [ 7.  8.  9.]]
a.sum(): 25.0
a.sum(axis=1): [ 9. 15. 24.]
a.sum(axis=0): [12. 15. 18.]
```

```
```
out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the
expected output but the type is cast if necessary.

ddof : int, optional

“Delta Degrees of Freedom”: the divisor used in calculation is N - ddof, where N
represents the number of elements. By default ddof is zero.

Returns

variance : ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise a reference to the
output array is returned.

See Also:

std

Standard deviation

mean

Average

numpy.doc.ufuncs

Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x.\text{mean}())^2) \).

The mean is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, ddof is specified, the
divisor \( N - ddof \) is used instead. In standard statistical practice, ddof=1 provides an unbiased estimator of
the variance of the infinite population. ddof=0 provides a maximum likelihood estimate of the variance for
normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and
nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the
input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a
higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a,0)
array([ 1., 1.])
>>> np.var(a,1)
array([ 0.25, 0.25])
```

In single precision, var() can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```
Computing the standard deviation in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

### Minimum/maximum

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.argmax(a[, axis, fill_value])</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.argmin(a[, axis, fill_value])</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.max(obj[, axis, out, fill_value])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.min(obj[, axis, out, fill_value])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.ptp(obj[, axis, out, fill_value])</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmax(self[, axis, fill_value, ...])</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmin(self[, axis, fill_value, ...])</code></td>
<td>Returns array of indices to the minimum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.max(self[, axis, out, fill_value])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.min(self[, axis, out, fill_value])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.ptp(self[, axis, out, fill_value])</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).</td>
</tr>
</tbody>
</table>

### argmax

Function version of the eponymous method.

#### Parameters

- **axis**: {None, integer}
  
  If None, the index is into the flattened array, otherwise along the specified axis

- **fill_value**: {var}, optional
  
  Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.

- **out**: {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

#### Returns

- **index_array**: {integer_array}

### Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```
max (obj, axis=None, out=None, fill_value=None)
Return the maximum along a given axis.

Parameters
axis : {None, int}, optional
Axis along which to operate. By default, axis is None and the flattened input is used.
out : array_like, optional
Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
fill_value : {var}, optional
Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns
amax : array_like
New array holding the result. If out was specified, out is returned.

See Also:
maximum_fill_value
Returns the maximum filling value for a given datatype.

min (obj, axis=None, out=None, fill_value=None)
Return the minimum along a given axis.

Parameters
axis : {None, int}, optional
Axis along which to operate. By default, axis is None and the flattened input is used.
out : array_like, optional
Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
fill_value : {var}, optional
Value used to fill in the masked values. If None, use the output of minimum_fill_value.

Returns
amin : array_like
New array holding the result. If out was specified, out is returned.

See Also:
minimum_fill_value
Returns the minimum filling value for a given datatype.

ptp (obj, axis=None, out=None, fill_value=None)
Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters
axis : {None, int}, optional
Axis along which to find the peaks. If None (default) the flattened array is used.
out : {None, array_like}, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**fill_value**: `{var}`, optional

Value used to fill in the masked values.

**Returns**

**ptp**: `ndarray`.

A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

**argmax** (`axis=None, fill_value=None, out=None`)

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value `fill_value`.

**Parameters**

- **axis**: `{None, integer}`
  
  If `None`, the index is into the flattened array, otherwise along the specified axis

- **fill_value**: `{var}`, optional
  
  Value used to fill in the masked values. If `None`, the output of `maximum_fill_value(self._data)` is used instead.

- **out**: `{None, array}`, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**index_array**: `{integer_array}`

**Examples**

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

**argmin** (`axis=None, fill_value=None, out=None`)

Return array of indices to the minimum values along the given axis.

**Parameters**

- **axis**: `{None, integer}`
  
  If `None`, the index is into the flattened array, otherwise along the specified axis

- **fill_value**: `{var}`, optional
  
  Value used to fill in the masked values. If `None`, the output of `minimum_fill_value(self._data)` is used instead.

- **out**: `{None, array}`, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.
Returns
{ndarray, scalar} :
If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

Examples

```python
>>> x = np.ma.array(arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> print x
[[-- --]
 [2 3]]
>>> print x.argmin(axis=0, fill_value=-1)
[0 0]
>>> print x.argmin(axis=0, fill_value=9)
[1 1]
```

`max` (axis=None, out=None, fill_value=None)
Return the maximum along a given axis.

Parameters
-axis : {None, int}, optional
 Axis along which to operate. By default, axis is None and the flattened input is used.
-out : array_like, optional
 Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
-fill_value : {var}, optional
 Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns
amax : array_like
New array holding the result. If out was specified, out is returned.

See Also:

`maximum_fill_value`
Returns the maximum filling value for a given datatype.

`min` (axis=None, out=None, fill_value=None)
Return the minimum along a given axis.

Parameters
-axis : {None, int}, optional
 Axis along which to operate. By default, axis is None and the flattened input is used.
-out : array_like, optional
 Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
-fill_value : {var}, optional
 Value used to fill in the masked values. If None, use the output of minimum_fill_value.
Returns

amin : array_like

New array holding the result. If out was specified, out is returned.

See Also:

minimum_fill_value

Returns the minimum filling value for a given datatype.

ptp (axis=None, out=None, fill_value=None)

Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters

axis : {None, int}, optional

Axis along which to find the peaks. If None (default) the flattened array is used.

out : {None, array_like}, optional

Alternative output array in which to place the result. It must have the same shape and
buffer length as the expected output but the type will be cast if necessary.

fill_value : {var}, optional

Value used to fill in the masked values.

Returns

ptp : ndarray.

A new array holding the result, unless out was specified, in which case a reference to
out is returned.

Sorting

ma.argsort (a[, axis, kind, order, ...])

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

ma.sort (a[, axis, kind, order, ...])

Return a sorted copy of an array.

ma.MaskedArray.argsort (self[, axis, fill_value, ...])

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

ma.MaskedArray.sort (self[, axis, kind, order, ...])

Return a sorted copy of an array.

argsort (a, axis=None, kind='quicksort', order=None, fill_value=None)

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

Parameters

axis : int, optional

Axis along which to sort. The default is -1 (last axis). If None, the flattened array is
used.

fill_value : var, optional

Value used to fill the array before sorting. The default is the fill_value attribute of the
input array.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm.

order : list, optional
When \(a\) is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**Returns**

- **index_array**: ndarray, int
  
  Array of indices that sort \(a\) along the specified axis. In other words, \(a[index_array]\) yields a sorted \(a\).

**See Also:**

- **sort**: Describes sorting algorithms used.
- **lexsort**: Indirect stable sort with multiple keys.
- **ndarray.sort**: Inplace sort.

**Notes**

See **sort** for notes on the different sorting algorithms.

**Examples**

```python
>>> a = np.ma.array([3, 2, 1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
              mask = [False False True],
              fill_value = 999999)
>>> a.argsort()
array([1, 0, 2])
```

**Parameters**

- **a**: array_like
  
  Array to be sorted.

- **axis**: int or None, optional
  
  Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

- **kind**: {'quicksort', 'mergesort', 'heapsort'}, optional
  
  Sorting algorithm. Default is ‘quicksort’.

- **order**: list, optional
  
  When \(a\) is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

- **endwith**: {True, False}, optional
  
  Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

- **fill_value**: {var}
Value used to fill in the masked values. If None, use the output of minimum_fill_value().

Returns

sorted_array : ndarray

Array of the same type and shape as a.

See Also:

ndarray.sort
Method to sort an array in-place.

argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in a sorted array.

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>~n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```python
>>> a = np.array([[1,4],[3,1]])
>>> np.sort(a) # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None) # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0) # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the order keyword to specify a field to use when sorting a structured array:

```python
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ...
   ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype) # create a structured array
>>> np.sort(a, order='height') # doctest: +SKIP
array([(1.7, 38), ('Arthur', 1.8, 41),
       ('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
       ('Lancelot', 1.8999999999999999, 38)],
      dtype=[('name', 'S10'), ('height', '<f8'), ('age', '<i4')])
```

Sort by age, then height if ages are equal:
```python
>>> np.sort(a, order=['age', 'height'])  # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
       ('Arthur', 1.8, 41)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

`argsort` *(axis=None, fill_value=None, kind='quicksort', order=None)*

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to `fill_value`.

**Parameters**

- `axis` : int, optional
  
  Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

- `fill_value` : var, optional
  
  Value used to fill the array before sorting. The default is the `fill_value` attribute of the input array.

- `kind` : {'quicksort', 'mergesort', 'heapsort'}, optional
  
  Sorting algorithm.

- `order` : list, optional
  
  When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**Returns**

- `index_array` : ndarray, int
  
  Array of indices that sort `a` along the specified axis. In other words, `a[index_array]` yields a sorted `a`.

**See Also:**

- `sort`
  
  Describes sorting algorithms used.

- `lexsort`
  
  Indirect stable sort with multiple keys.

- `ndarray.sort`
  
  Inplace sort.

**Notes**

See `sort` for notes on the different sorting algorithms.

**Examples**

```python
>>> a = np.ma.array([3, 2, 1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
             mask = [False False True],
            fill_value = 999999)
>>> a.argsort()
array([1, 0, 2])
```
sort (axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)
Return a sorted copy of an array.

Parameters

a : array_like
   Array to be sorted.

axis : int or None, optional
   Axis along which to sort. If None, the array is flattened before sorting. The default is
   -1, which sorts along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
   Sorting algorithm. Default is 'quicksort'.

order : list, optional
   When a is a structured array, this argument specifies which fields to compare first, second,
   and so on. This list does not need to include all of the fields.

d, : {True, False}, optional
   Whether missing values (if any) should be forced in the upper indices (at the end of the
   array) (True) or lower indices (at the beginning).

fill_value : {var}
   Value used to fill in the masked values. If None, use the the output of minimum_fill_value().

Returns

sorted_array : ndarray
   Array of the same type and shape as a.

See Also:

ndarray.sort
   Method to sort an array in-place.

argsort
   Indirect sort.

lexsort
   Indirect stable sort on multiple keys.

searchsorted
   Find elements in a sorted array.

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space
size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The
three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>-n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently,
sorting along the last axis is faster and uses less space than sorting along any other axis.
Examples

```python
>>> a = np.array([[1, 4], [3, 1]])
>>> np.sort(a)                      # sort along the last axis
array([[1, 4],
       [3, 1]])
>>> np.sort(a, axis=None)           # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)              # sort along the first axis
array([[1, 1],
       [3, 4]])

Use the `order` keyword to specify a field to use when sorting a structured array:

```python
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ...
          ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)    # create a structured array
>>> np.sort(a, order='height')           # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
       ('Lancelot', 1.9, 38)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

Sort by age, then height if ages are equal:

```python
>>> np.sort(a, order=['age', 'height'])  # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Lancelot', 1.9, 38),
       ('Arthur', 1.8, 41)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

Algebra

```
m.diag(v[, k])
m.dot(a, b[, strict])
m.identity(n[, dtype])
m.inner(a, b)
m.innerproduct(a, b)
m.outer(a, b)
m.outerproduct(a, b)
m.trace(self[, offset, axis1, axis2, ...])
m.transpose(a[, axes])
ma.MaskedArray.trace ([offset, axis1, axis2, ...])
ma.MaskedArray.transpose (*axes)
```

Extract a diagonal or construct a diagonal array.
Return the dot product of two arrays.
Return the identity array.
Inner product of two arrays.
Inner product of two arrays.
Compute the outer product of two vectors.
Compute the outer product of two vectors.
Return the sum along diagonals of the array.
Permute the dimensions of an array.
Return the sum along diagonals of the array.
Returns a view of the array with axes transposed.

```
diag(v; k=0)
```

Extract a diagonal or construct a diagonal array.

This function is the equivalent of `numpy.diag` that takes masked values into account, see `numpy.diag` for details.

See Also:

```
numpy.diag
```

Equivalent function for ndarrays.
**dot** *(a, b, strict=False)*

Return the dot product of two arrays.

**Note:** Works only with 2-D arrays at the moment.

This function is the equivalent of `numpy.dot` that takes masked values into account, see `numpy.dot` for details.

**Parameters**

- **a, b:** ndarray
  
  Inputs arrays.
  
- **strict:** bool, optional
  
  Whether masked data are propagated (True) or set to 0 (False) for the computation. Default is False. Propagating the mask means that if a masked value appears in a row or column, the whole row or column is considered masked.

**See Also:**

- `numpy.dot`
  
  Equivalent function for ndarrays.

**Examples**

```python
dot = dot(a, b)
dot = np.ma.dot(a, b)
```

**identity()**

Return the identity array.

The identity array is a square array with ones on the main diagonal.

**Parameters**

- **n:** int
  
  Number of rows (and columns) in n x n output.

- **dtype:** data-type, optional
  
  Data-type of the output. Defaults to float.

**Returns**

- **out:** ndarray
\( n \times n \) array with its main diagonal set to one, and all other elements 0.

**Examples**

```python
>>> np.identity(3)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

**inner** \((a, b)\)

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

- \( a, b \) : array_like

  If \( a \) and \( b \) are nonscalar, their last dimensions of must match.

**Returns**

- \( out \) : ndarray

  \( out.shape = a.shape[:-1] + b.shape[:-1] \)

**Raises**

- \( ValueError \)

  If the last dimension of \( a \) and \( b \) has different size.

**See Also:**

- **tensordot**
  
  Sum products over arbitrary axes.

- **dot**
  
  Generalised matrix product, using second last dimension of \( b \).

**Notes**

Masked values are replaced by 0.

**Examples**

Ordinary inner product for vectors:

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([0, 1, 0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2, 3, 4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[14, 38, 62],
       [86, 110, 134]])
```

An example where \( b \) is a scalar:
```python
code
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

**innerproduct** (*a*, *b*)
Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

- **a**, **b**: array_like
  If *a* and *b* are nonscalar, their last dimensions of must match.

**Returns**

- **out**: ndarray
  out.shape = a.shape[:-1] + b.shape[:-1]

**Raises**

- **ValueError**: If the last dimension of *a* and *b* has different size.

**See Also:**

tensordot
  Sum products over arbitrary axes.

dot
  Generalised matrix product, using second last dimension of *b*.

**Notes**

Masked values are replaced by 0.

**Examples**

Ordinary inner product for vectors:

```python
code
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
code
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```python
code
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```
outer \((a, b)\)

Compute the outer product of two vectors.

Given two vectors, \(a = [a_0, \ a_1, \ldots, \ a_M]\) and \(b = [b_0, \ b_1, \ldots, \ b_N]\), the outer product \([R57]\) is:

\[
\begin{bmatrix}
a_0 \times b_0 & a_0 \times b_1 & \cdots & a_0 \times b_N \\
a_1 \times b_0 & \cdots & \cdots & \cdots \\
\vdots & \ddots & \ddots & \ddots \\
a_M \times b_0 & \cdots & \cdots & \cdots \\
\end{bmatrix}
\]

Parameters

- \(a, b\) : array_like, shape (M,), (N,)

  First and second input vectors. Inputs are flattened if they are not already 1-dimensional.

Returns

- \(out\) : ndarray, shape (M, N)

  \(out[i, j] = a[i] \times b[j]\)

Notes

Masked values are replaced by 0.

References

\([R57]\)

Examples

Make a (very coarse) grid for computing a Mandelbrot set:

```python
>>> r1 = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> r1
array([[-2., -1., 0., 1., 2.],
       [-2., -1., 0., 1., 2.],
       [-2., -1., 0., 1., 2.],
       [-2., -1., 0., 1., 2.],
       [-2., -1., 0., 1., 2.]])
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
       [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
       [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
       [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
>>> grid = r1 + im
```

An example using a “vector” of letters:
outerproduct (a, b)
Compute the outer product of two vectors.

Given two vectors, \(a = [a_0, a_1, \ldots, a_M]\) and \(b = [b_0, b_1, \ldots, b_N]\), the outer product \([R58]\) is:

\[
[a_0*b_0 \quad a_0*b_1 \quad \ldots \quad a_0*b_N ]
\]
\[
[a_1*b_0 \quad \ldots \quad \ldots \quad \ldots \quad a_1*b_N ]
\]
\[
[ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots ]
\]
\[
[a_M*b_0 \quad a_M*b_1 \quad \ldots \quad a_M*b_N ]
\]

Parameters

a, b : array_like, shape (M,), (N,)

First and second input vectors. Inputs are flattened if they are not already 1-dimensional.

Returns

out : ndarray, shape (M, N)

out[i, j] = a[i] * b[j]

Notes

Masked values are replaced by 0.

References

[R58]

Examples

Make a (very coarse) grid for computing a Mandelbrot set:

>>> r1 = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> r1
array([[[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]]))

>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])

>>> grid = r1 + im
>>> grid
array([[ 0.+2.j, 0.+3.j, 0.+4.j, 0.+5.j, 0.+6.j],
        [ 0.+1.j, 0.+2.j, 0.+3.j, 0.+4.j, 0.+5.j],
        [ 0.+0.j, 0.+1.j, 0.+2.j, 0.+3.j, 0.+4.j],
        [ 0.-1.j, 0.-2.j, 0.-3.j, 0.-4.j, 0.-5.j],
        [ 0.-2.j, 0.-3.j, 0.-4.j, 0.-5.j, 0.-6.j]])
An example using a “vector” of letters:

```python
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
       [b, bb, bbb],
       [c, cc, ccc]], dtype=object)
```

trace()

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

- `numpy.trace`
  equivalent function

transpose(a, axes=None)

Permute the dimensions of an array.

This function is exactly equivalent to `numpy.transpose`.

See Also:

- `numpy.transpose`
  Equivalent function in top-level NumPy module.

Examples

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[ 0  1]
 [ 2 --]],
mask =
[[False False]
 [False  True]],
fill_value = 999999)
```

```python
>>> ma.transpose(x)
masked_array(data =
[[ 0  2]
 [ 1 --]],
mask =
[[False False]
 [False  True]],
fill_value = 999999)
```

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()**

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]])
```

**Polynomial fit**

- **ma.vander** `(x[, n])` Generate a Van der Monde matrix.
- **ma.polyfit** `(x, y, deg[, rcond, full])` Least squares polynomial fit.
- **vander** `(x, n=none)` Generate a Van der Monde matrix.
The columns of the output matrix are decreasing powers of the input vector. Specifically, the i-th output column is the input vector to the power of \( N - i - 1 \). Such a matrix with a geometric progression in each row is named Van Der Monde, or Vandermonde matrix, from Alexandre-Theophile Vandermonde.

**Parameters**

- **x**: array_like
  - 1-D input array.

- **N**: int, optional
  - Order of (number of columns in) the output. If \( N \) is not specified, a square array is returned (\( N = \text{len}(x) \)).

**Returns**

- **out**: ndarray
  - Van der Monde matrix of order \( N \). The first column is \( x^{(N-1)} \), the second \( x^{(N-2)} \) and so forth.

**Notes**

Masked values in the input array result in rows of zeros.

**References**

[R61]

**Examples**

```python
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```python
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```python
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[ 1,  1,  1,  1],
       [ 8,  4,  2,  1],
       [27,  9,  3,  1],
       [125, 25,  5,  1]])
```

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```python
>>> np.linalg.det(np.vander(x))
48.000000000000043
```

```python
>>> (5-3) * (5-2) * (5-1) * (3-2) * (3-1) * (2-1)
48
```
polyfit(x, y, deg, rcond=None, full=False)

Least squares polynomial fit.

Fit a polynomial \( p(x) = p[0] \times x^{\deg} + \ldots + p[\deg] \) of degree \( \deg \) to points \((x, y)\). Returns a vector of coefficients \( p \) that minimises the squared error.

**Parameters**

- **x**: array_like, shape (M,)
  - x-coordinates of the M sample points \((x[i], y[i])\).

- **y**: array_like, shape (M,) or (M, K)
  - y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- **deg**: int
  - Degree of the fitting polynomial

- **rcond**: float, optional
  - Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \( \text{len}(x) \times \text{eps} \), where \( \text{eps} \) is the relative precision of the float type, about 2e-16 in most cases.

- **full**: bool, optional
  - Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**Returns**

- **p**: ndarray, shape (M,) or (M, K)
  - Polynomial coefficients, highest power first. If \( y \) was 2-D, the coefficients for \( k \)-th data set are in \( p[:, k] \).

- **residuals**, **rank**, **singular_values**, **rcond**: present only if **full** = True
  - Residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of **rcond**. For more details, see `linalg.lstsq`.

**See Also:**

- **polyval**: Computes polynomial values.
- **linalg.lstsq**: Computes a least-squares fit.
- **scipy.interpolate.UnivariateSpline**: Computes spline fits.

**Notes**

Any masked values in \( x \) is propagated in \( y \), and vice-versa.

**References**

[R59], [R60]
Examples

```python
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206, 1.69312169, -0.03968254])
```

It is convenient to use `poly1d` objects for dealing with polynomials:

```python
>>> p = np.poly1d(z)
```

```python
>>> p(0.5)
0.6143849206349179
```

```python
>>> p(3.5)
-0.34732142857143039
```

```python
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```python
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
```

```python
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
```

```python
>>> p30(5)
-0.99999999999999445
```

```python
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```python
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
```

```python
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
```

```python
>>> plt.ylim(-2,2)
```

```python
>>> plt.show()
```

Clipping and rounding

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.around()</code></td>
<td>Round an array to the given number of decimals.</td>
</tr>
<tr>
<td><code>ma.clip(a, a_min, a_max[, out])</code></td>
<td>Clip (limit) the values in an array.</td>
</tr>
<tr>
<td><code>ma.round(a[, decimals, out])</code></td>
<td>Return a copy of a, rounded to ‘decimals’ places.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.clip(a_min, a_max[, out])</code></td>
<td>Return an array whose values are limited to <code>[a_min, a_max]</code>.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.round([decimals, out])</code></td>
<td>Return an array rounded a to the given number of decimals.</td>
</tr>
</tbody>
</table>

**around()**

Round an array to the given number of decimals.

Refer to `around` for full documentation.

See Also:

around

equivalent function

**clip(a, a_min, a_max, out=None)**

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters

- **a** : array_like
  Array containing elements to clip.

- **a_min** : scalar or array_like
  Minimum value.

- **a_max** : scalar or array_like
  Maximum value. If `a_min` or `a_max` are array_like, then they will be broadcasted to the shape of `a`.

- **out** : ndarray, optional
  The results will be placed in this array. It may be the input array for in-place clipping. `out` must be of the right shape to hold the output. Its type is preserved.

Returns

- **clipped_array** : ndarray
  An array with the elements of `a`, but where values `< a_min` are replaced with `a_min`, and those `> a_max` with `a_max`.

See Also:

numpy.doc.ufuncs

Section “Output arguments”

**Examples**

```python
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
```

1.6. Masked arrays
array([3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a
array([3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])

round(a, decimals=0, out=None)
Return a copy of a, rounded to ‘decimals’ places.

When ‘decimals’ is negative, it specifies the number of positions to the left of the decimal point. The real and imaginary parts of complex numbers are rounded separately. Nothing is done if the array is not of float type and ‘decimals’ is greater than or equal to 0.

Parameters

decimals : int
    Number of decimals to round to. May be negative.

out : array_like
    Existing array to use for output. If not given, returns a default copy of a.

Notes
If out is given and does not have a mask attribute, the mask of a is lost!

clip()
Return an array whose values are limited to [a_min, a_max].

Refer to numpy.clip for full documentation.

See Also:

numpy.clip
equivalent function

round(decimals=0, out=None)
Return an array rounded a to the given number of decimals.

Refer to numpy.around for full documentation.

See Also:

numpy.around
equivalent function

Miscellanea

ma.allequal(a, b[, fill_value])
Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

ma.allclose(a, b[, masked_equal, rtol, ...])
Return True if two arrays are element-wise equal within a tolerance.

ma.apply_along_axis(func1d, axis, arr, ...)
Apply a function to 1-D slices along the given axis.

ma.arange([start[, stop[, step[, ...]...]]]
Return evenly spaced values within a given interval.

Use an index array to construct a new array from a set of choices.

ma.ediff1d(arr[, to_end, to_begin])
Compute the differences between consecutive elements of an array.

ma.indices(dimensions[, dtype])
Return an array representing the indices of a grid.

ma.where(condition[, x, y])
Return a masked array with elements from x or y, depending on condition.
allequal\((a, b, \text{fill_value}=\text{True})\)

Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

**Parameters**

- a, b : array_like
  
  Input arrays to compare.

- fill_value : bool, optional
  
  Whether masked values in a or b are considered equal (True) or not (False).

**Returns**

- y : bool
  
  Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

**See Also:**

all, any, numpy.ma.allclose

**Examples**

```python
>>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data = [10000000000.0 1e-07 --],
              mask = [False False True],
              fill_value=1e+20)

>>> b = array([1e10, 1e-7, -42.0])
>>> b
array([ 1.00000000e+10, 1.00000000e-07, -4.20000000e+01])

>>> ma.allequal(a, b, fill_value=False)
False

>>> ma.allequal(a, b)
True
```

allclose\((a, b, \text{masked_equal}=\text{True}, \text{rtol}=1.0000000000000001e-05, \text{atol}=1e-08, \text{fill_value}=\text{None})\)

Returns True if two arrays are element-wise equal within a tolerance.

This function is equivalent to allclose except that masked values are treated as equal (default) or unequal, depending on the masked_equal argument.

**Parameters**

- a, b : array_like
  
  Input arrays to compare.

- masked_equal : bool, optional
  
  Whether masked values in a and b are considered equal (True) or not (False). They are considered equal by default.

- rtol : float, optional
  
  Relative tolerance. The relative difference is equal to \(\text{rtol} \times b\). Default is 1e-5.

- atol : float, optional
  
  Absolute tolerance. The absolute difference is equal to \(\text{atol}\). Default is 1e-8.

- fill_value : bool, optional

1.6. Masked arrays
**Deprecation** - Whether masked values in a or b are considered equal (True) or not (False).

Returns

y : bool

Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

See Also:

call, any

**numpy.allclose**
the non-masked allclose.

Notes

If the following equation is element-wise True, then allclose returns True:

\[
\text{absolute}(a - b) \leq \text{\texttt{'atol'} + \text{\texttt{'rtol'}} \times \text{absolute}(b)}
\]

Return True if all elements of a and b are equal subject to given tolerances.

Examples

```python
>>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data = [10000000000.0 1e-07 --],
mask = [False False True], fill_value = 1e+20)
>>> b = ma.array([1e10, 1e-8, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
False

>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

Masked values are not compared directly.

```python
>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, 42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

**apply_along_axis(func1d, axis, arr, *args, **kwargs)**

Apply a function to 1-D slices along the given axis.

Execute `func1d(a, *args)` where `func1d` operates on 1-D arrays and `a` is a 1-D slice of `arr` along `axis`.

Parameters

func1d : function
This function should accept 1-D arrays. It is applied to 1-D slices of \textit{arr} along the specified axis.

\textbf{axis} : integer

Axis along which \textit{arr} is sliced.

\textbf{arr} : ndarray

Input array.

\textbf{args} : any

Additional arguments to \textit{func1d}.

\textbf{Returns}

\textbf{outarr} : ndarray

The output array. The shape of \textit{outarr} is identical to the shape of \textit{arr}, except along the \textit{axis} dimension, where the length of \textit{outarr} is equal to the size of the return value of \textit{func1d}. If \textit{func1d} returns a scalar \textit{outarr} will have one fewer dimensions than \textit{arr}.

\textbf{See Also:}

\textit{apply_over_axes}

Apply a function repeatedly over multiple axes.

\textbf{Examples}

>>> def my_func(a):
...     """Average first and last element of a 1-D array""
...     return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2., 5., 8.])

For a function that doesn’t return a scalar, the number of dimensions in \textit{outarr} is the same as \textit{arr}.

>>> def new_func(a):
...     """Divide elements of a by 2.""
...     return a * 0.5
>>> b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> np.apply_along_axis(new_func, 0, b)
array([[ 0.5, 1. , 1.5],
       [ 2. , 2.5, 3. ],
       [ 3.5, 4. , 4.5]])

\textbf{arange()}

Return evenly spaced values within a given interval.

Values are generated within the half-open interval \textit{[start, stop)} (in other words, the interval including \textit{start} but excluding \textit{stop}). For integer arguments the function is equivalent to the Python built-in \textit{range} function, but returns a \textit{ndarray} rather than a list.

\textbf{Parameters}

\textbf{start} : number, optional

Start of interval. The interval includes this value. The default start value is 0.

\textbf{stop} : number

1.6. Masked arrays
End of interval. The interval does not include this value.

**step**: number, optional

Spacing between values. For any output `out`, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If `step` is specified, `start` must also be given.

**dtype**: dtype

The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

**Returns**

**out**: ndarray

Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of `out` being greater than `stop`.

**See Also:**

**linspace**

Evenly spaced numbers with careful handling of endpoints.

**ogrid**

Arrays of evenly spaced numbers in N-dimensions

**mgrid**

Grid-shaped arrays of evenly spaced numbers in N-dimensions

**Examples**

```python
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0., 1., 2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

**choose** *(indices, choices, out=None, mode='raise')*

Use an index array to construct a new array from a set of choices.

Given an array of integers and a set of n choice arrays, this method will create a new array that merges each of the choice arrays. Where a value in `a` is i, the new array will have the value that `choices[i]` contains in the same place.

**Parameters**

- **a**: ndarray of ints
  
  This array must contain integers in `[0, n-1]`, where n is the number of choices.

- **choices**: sequence of arrays
  
  Choice arrays. The index array and all of the choices should be broadcastable to the same shape.

- **out**: array, optional
If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave.
- ‘raise’: raise an error
- ‘wrap’: wrap around
- ‘clip’: clip to the range

Returns
merged_array : array

See Also:
choose equivalent function

Examples

```python
>>> choice = np.array([[1,1,1], [2,2,2], [3,3,3]])
>>> a = np.array([2, 1, 0])
>>> np.ma.choose(a, choice)
masked_array(data = [3 2 1],
mask = False,
fill_value=999999)
```

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

Compute the differences between consecutive elements of an array.

This function is the equivalent of numpy.ediff1d that takes masked values into account, see numpy.ediff1d for details.

See Also:

numpy.ediff1d

Equivalent function for ndarrays.

indices(dimensions, dtype=<type 'int'>)

Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

Parameters
- dimensions : sequence of ints
  The shape of the grid.
- dtype : dtype, optional
  Data type of the result.

Returns
grid : ndarray
  The array of grid indices, grid.shape = (len(dimensions),) + tuple(dimensions).
See Also:

mgrid, meshgrid

Notes

The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if `dimensions` is a tuple \((r_0, \ldots, r_{N-1})\) of length \(N\), the output shape is \((N, r_0, \ldots, r_{N-1})\).

The subarrays `grid[k]` contains the N-D array of indices along the \(k\)-th axis. Explicitly:

\[
grid[k, i_0, i_1, \ldots, i_{N-1}] = ik
\]

Examples

```python
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]  # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]  # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```python
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```

Note that it would be more straightforward in the above example to extract the required elements directly with `x[::2, ::3].`  

`where(condition, x=None, y=None)`

Return a masked array with elements from \(x\) or \(y\), depending on condition.

Returns a masked array, shaped like condition, where the elements are from \(x\) when `condition` is True, and from \(y\) otherwise. If neither \(x\) nor \(y\) are given, the function returns a tuple of indices where `condition` is True (the result of `condition.nonzero()`).

Parameters

- `condition` : array_like, bool
  The condition to meet. For each True element, yield the corresponding element from \(x\), otherwise from \(y\).

- `x, y` : array_like, optional
  Values from which to choose. \(x\) and \(y\) need to have the same shape as condition, or be broadcast-able to that shape.

Returns

- `out` : MaskedArray or tuple of ndarrays
  The resulting masked array if \(x\) and \(y\) were given, otherwise the result of `condition.nonzero()`.

See Also:
**numpy.where**

Equivalent function in the top-level NumPy module.

**Examples**

```python
>>> x = np.ma.array(np.arange(9.).reshape(3, 3), mask=[[0, 1, 0],
... [1, 0, 1],
... [0, 1, 0]])
>>> print x
[[0.0 -- 2.0]
[-- 4.0 --]
[6.0 -- 8.0]]
>>> np.ma.where(x > 5)  # return the indices where x > 5
(array([2, 2]), array([0, 2]))

>>> print np.ma.where(x > 5, x, -3.1416)
[[-3.1416 -- -3.1416]
[-- -3.1416 --]
[6.0 -- 8.0]]
```

### 1.7 The Array Interface

**Warning:** This page describes the old, deprecated array interface. Everything still works as described as of numpy 1.2 and on into the foreseeable future, but new development should target [PEP 3118 – The Revised Buffer Protocol](https://www.python.org/dev/peps/pep-0318/). PEP 3118 was incorporated into Python 2.6 and 3.0, and is additionally supported by Cython’s `numpy` buffer support. (See the Cython `numpy` tutorial.) Cython provides a way to write code that supports the buffer protocol with Python versions older than 2.6 because it has a backward-compatible implementation utilizing the legacy array interface described here.

#### version

3

The array interface (sometimes called array protocol) was created in 2005 as a means for array-like Python objects to re-use each other’s data buffers intelligently whenever possible. The homogeneous N-dimensional array interface is a default mechanism for objects to share N-dimensional array memory and information. The interface consists of a Python-side and a C-side using two attributes. Objects wishing to be considered an N-dimensional array in application code should support at least one of these attributes. Objects wishing to support an N-dimensional array in application code should look for at least one of these attributes and use the information provided appropriately.

This interface describes homogeneous arrays in the sense that each item of the array has the same “type”. This type can be very simple or it can be a quite arbitrary and complicated C-like structure.

There are two ways to use the interface: A Python side and a C-side. Both are separate attributes.

#### 1.7.1 Python side

This approach to the interface consists of the object having an `__array_interface__` attribute.

```python
__array_interface__
```

A dictionary of items (3 required and 5 optional). The optional keys in the dictionary have implied defaults if they are not provided.

The keys are:
shape (required)

Tuple whose elements are the array size in each dimension. Each entry is an integer (a Python int or long). Note that these integers could be larger than the platform “int” or “long” could hold (a Python int is a C long). It is up to the code using this attribute to handle this appropriately; either by raising an error when overflow is possible, or by using Py_LONG_LONG as the C type for the shapes.

typestr (required)

A string providing the basic type of the homogenous array. The basic string format consists of 3 parts: a character describing the byteorder of the data (<: little-endian, >: big-endian, |: not-relevant), a character code giving the basic type of the array, and an integer providing the number of bytes the type uses.

The basic type character codes are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Bit field (following integer gives the number of bits in the bit field).</td>
</tr>
<tr>
<td>b</td>
<td>Boolean (integer type where all values are only True or False)</td>
</tr>
<tr>
<td>i</td>
<td>Integer</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>f</td>
<td>Floating point</td>
</tr>
<tr>
<td>c</td>
<td>Complex floating point</td>
</tr>
<tr>
<td>o</td>
<td>Object (i.e. the memory contains a pointer to PyObject)</td>
</tr>
<tr>
<td>s</td>
<td>String (fixed-length sequence of char)</td>
</tr>
<tr>
<td>u</td>
<td>Unicode (fixed-length sequence of Py_UNICODE)</td>
</tr>
<tr>
<td>v</td>
<td>Other (void * – each item is a fixed-size chunk of memory)</td>
</tr>
</tbody>
</table>

descr (optional)

A list of tuples providing a more detailed description of the memory layout for each item in the homogeneous array. Each tuple in the list has two or three elements. Normally, this attribute would be used when typestr is V[0-9]+, but this is not a requirement. The only requirement is that the number of bytes represented in the typestr key is the same as the total number of bytes represented here. The idea is to support descriptions of C-like structs (records) that make up array elements. The elements of each tuple in the list are

1. A string providing a name associated with this portion of the record. This could also be a tuple of (‘full name’, ‘basic name’) where basic name would be a valid Python variable name representing the full name of the field.

2. Either a basic-type description string as in typestr or another list (for nested records)

3. An optional shape tuple providing how many times this part of the record should be repeated. No repeats are assumed if this is not given. Very complicated structures can be described using this generic interface. Notice, however, that each element of the array is still of the same data-type. Some examples of using this interface are given below.

Default: ["", typestr]

data (optional)

A 2-tuple whose first argument is an integer (a long integer if necessary) that points to the data-area storing the array contents. This pointer must point to the first element of data (in other words any offset is always ignored in this case). The second entry in the tuple is a read-only flag (true means the data area is read-only).

This attribute can also be an object exposing the buffer interface which will be used to share the data. If this key is not present (or returns None), then memory sharing will be done through the buffer interface of the object itself. In this case, the offset key can be used to indicate the start of the buffer. A reference to the object exposing the array interface must be stored by the new object if the memory area is to be secured.
Default: None

strides (optional)

Either None to indicate a C-style contiguous array or a Tuple of strides which provides the number of bytes needed to jump to the next array element in the corresponding dimension. Each entry must be an integer (a Python int or long). As with shape, the values may be larger than can be represented by a C “int” or “long”; the calling code should handle this appropriately, either by raising an error, or by using Py_LONG_LONG in C. The default is None which implies a C-style contiguous memory buffer. In this model, the last dimension of the array varies the fastest. For example, the default strides tuple for an object whose array entries are 8 bytes long and whose shape is (10,20,30) would be (4800, 240, 8)

Default: None (C-style contiguous)

mask (optional)

None or an object exposing the array interface. All elements of the mask array should be interpreted only as true or not true indicating which elements of this array are valid. The shape of this object should be “broadcastable” to the shape of the original array.

Default: None (All array values are valid)

offset (optional)

An integer offset into the array data region. This can only be used when data is None or returns a buffer object.

Default: 0.

version (required)

An integer showing the version of the interface (i.e. 3 for this version). Be careful not to use this to invalidate objects exposing future versions of the interface.

1.7.2 C-struct access

This approach to the array interface allows for faster access to an array using only one attribute lookup and a well-defined C-structure.

__array_struct__

A PyObject whose voidptr member contains a pointer to a filled PyArrayInterface structure. Memory for the structure is dynamically created and the PyObject is also created with an appropriate destructor so the retriever of this attribute simply has to apply Py_DECREF to the object returned by this attribute when it is finished. Also, either the data needs to be copied out, or a reference to the object exposing this attribute must be held to ensure the data is not freed. Objects exposing the __array_struct__ interface must also not reallocate their memory if other objects are referencing them.

The PyArrayInterface structure is defined in numpy/ndarrayobject.h as:

typedef struct {
    int two; /* contains the integer 2 -- simple sanity check */
    int nd; /* number of dimensions */
    char typekind; /* kind in array --- character code of typestr */
    int itemsize; /* size of each element */
    int flags; /* flags indicating how the data should be interpreted */
    Py_intptr_t *shape; /* A length-nd array of shape information */
    Py_intptr_t *strides; /* A length-nd array of stride information */
    void *data; /* A pointer to the first element of the array */
} /* end of array_struct */

1.7. The Array Interface
PyObject *descr;  /* NULL or data-description (same as descr key of __array_interface__) -- must set ARR_HAS_DESCR flag or this will be ignored. */

PyArrayInterface;

The flags member may consist of 5 bits showing how the data should be interpreted and one bit showing how the Interface should be interpreted. The data-bits are CONTIGUOUS (0x1), FORTRAN (0x2), ALIGNED (0x100), NOTSWAPPED (0x200), and WRITEABLE (0x400). A final flag ARR_HAS_DESCR (0x800) indicates whether or not this structure has the arrdescr field. The field should not be accessed unless this flag is present.

New since June 16, 2006:

In the past most implementations used the “desc” member of the PyCObject itself (do not confuse this with the “descr” member of the PyArrayInterface structure above — they are two separate things) to hold the pointer to the object exposing the interface. This is now an explicit part of the interface. Be sure to own a reference to the object when the PyCObject is created using PyCObject_FromVoidPtrAndDesc.

1.7.3 Type description examples

For clarity it is useful to provide some examples of the type description and corresponding __array_interface__ ‘descr’ entries. Thanks to Scott Gilbert for these examples:

In every case, the ‘descr’ key is optional, but of course provides more information which may be important for various applications:

* Float data
  
<table>
<thead>
<tr>
<th>typestr</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘&gt;f4’</td>
<td>[('', ‘&gt;f4’)]</td>
</tr>
</tbody>
</table>

* Complex double
  
<table>
<thead>
<tr>
<th>typestr</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘&gt;c8’</td>
<td>[‘real’, ‘&gt;f4’], (‘imag’, ‘&gt;f4’)]</td>
</tr>
</tbody>
</table>

* RGB Pixel data
  
<table>
<thead>
<tr>
<th>typestr</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘</td>
<td>V3’</td>
</tr>
</tbody>
</table>

* Mixed endian (weird but could happen).
  
<table>
<thead>
<tr>
<th>typestr</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘</td>
<td>V8’ (or ‘&lt;u8’)</td>
</tr>
</tbody>
</table>

* Nested structure
  
<table>
<thead>
<tr>
<th>typestr</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘</td>
<td>V8’ (or ‘&lt;u8’ if you want)</td>
</tr>
</tbody>
</table>

* Nested array
  
<table>
<thead>
<tr>
<th>typestr</th>
<th>descr</th>
</tr>
</thead>
</table>

Chapter 1. Array objects
It should be clear that any record type could be described using this interface.

### 1.7.4 Differences with Array interface (Version 2)

The version 2 interface was very similar. The differences were largely aesthetic. In particular:

1. The PyArrayInterface structure had no descr member at the end (and therefore no flag ARR_HAS_DESCR).
2. The desc member of the PyCObject returned from __array_struct__ was not specified. Usually, it was the object exposing the array (so that a reference to it could be kept and destroyed when the C-object was destroyed). Now it must be a tuple whose first element is a string with “PyArrayInterface Version #” and whose second element is the object exposing the array.
3. The tuple returned from __array_interface__['data'] used to be a hex-string (now it is an integer or a long integer).
4. There was no __array_interface__ attribute instead all of the keys (except for version) in the __array_interface__ dictionary were their own attribute: Thus to obtain the Python-side information you had to access separately the attributes:
   - __array_data__
   - __array_shape__
   - __array_strides__
   - __array_typestr__
   - __array_descr__
   - __array_offset__
   - __array_mask__
A universal function (or *ufunc* for short) is a function that operates on *ndarrays* in an element-by-element fashion, supporting *array broadcasting*, *type casting*, and several other standard features. That is, a *ufunc* is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.

In Numpy, universal functions are instances of the `numpy.ufunc` class. Many of the built-in functions are implemented in compiled C code, but *ufunc* instances can also be produced using the `frompyfunc` factory function.

### 2.1 Broadcasting

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs. Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on. Broadcasting can be understood by four rules:

1. All input arrays with `ndim` smaller than the input array of largest `ndim`, have 1’s prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the *ufunc* will simply not step along that dimension (the `stride` will be 0 for that dimension).

Broadcasting is used throughout NumPy to decide how to handle disparately shaped arrays; for example, all arithmetic operations (+, −, ×, ...) between *ndarrays* broadcast the arrays before operation. A set of arrays is called “broadcastable” to the same shape if the above rules produce a valid result, *i.e.*, one of the following is true:

1. The arrays all have exactly the same shape.
2. The arrays all have the same number of dimensions and the length of each dimensions is either a common length or 1.
3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

**Example**

If `a.shape` is (5,1), `b.shape` is (1,6), `c.shape` is (6,) and `d.shape` is () so that `d` is a scalar, then `a`, `b`, `c`, and `d` are all broadcastable to dimension (5,6); and

- `a` acts like a (5,6) array where `a[:,0]` is broadcast to the other columns,
- `b` acts like a (5,6) array where `b[0,:]` is broadcast to the other rows,
• *c* acts like a (1,6) array and therefore like a (5,6) array where *c[:]* is broadcast to every row, and finally,

• *d* acts like a (5,6) array where the single value is repeated.

### 2.2 Output type determination

The output of the ufunc (and its methods) is not necessarily an `ndarray`, if all input arguments are not `ndarrays`.

All output arrays will be passed to the `__array_prepare__` and `__array_wrap__` methods of the input (besides `ndarrays`, and scalars) that defines it and has the highest `__array_priority__` of any other input to the universal function. The default `__array_priority__` of the `ndarray` is 0.0, and the default `__array_priority__` of a subtype is 1.0. Matrices have `__array_priority__` equal to 10.0.

All ufuncs can also take output arguments. If necessary, output will be cast to the data-type(s) of the provided output array(s). If a class with an `__array__` method is used for the output, results will be written to the object returned by `__array__`. Then, if the class also has an `__array_prepare__` method, it is called so metadata may be determined based on the context of the ufunc (the context consisting of the ufunc itself, the arguments passed to the ufunc, and the ufunc domain.) The array object returned by `__array_prepare__` is passed to the ufunc for computation. Finally, if the class also has an `__array_wrap__` method, the returned `ndarray` result will be passed to that method just before passing control back to the caller.

### 2.3 Use of internal buffers

Internally, buffers are used for misaligned data, swapped data, and data that has to be converted from one data type to another. The size of internal buffers is settable on a per-thread basis. There can be up to \(2(n_{\text{inputs}} + n_{\text{outputs}})\) buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of a buffer is 10,000 elements. Whenever buffer-based calculation would be needed, but all input arrays are smaller than the buffer size, those misbehaved or incorrectly-typed arrays will be copied before the calculation proceeds. Adjusting the size of the buffer may therefore alter the speed at which ufunc calculations of various sorts are completed. A simple interface for setting this variable is accessible using the function

```python
setbufsize(size)
```

Set the size of the buffer used in ufuncs.

**Parameters**

- `size` : int
  
  Size of buffer.

### 2.4 Error handling

Universal functions can trip special floating-point status registers in your hardware (such as divide-by-zero). If available on your platform, these registers will be regularly checked during calculation. Error handling is controlled on a per-thread basis, and can be configured using the functions

```python
seterr([all, divide, over, ...])
seterrcall(func)
```

Set how floating-point errors are handled.

```python
seterr(all=None, divide=None, over=None, under=None, invalid=None)
```

Set the floating-point error callback function or log object.
Note that operations on integer scalar types (such as `int16`) are handled like floating point, and are affected by these settings.

**Parameters**

- `all`: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  
  Set treatment for all types of floating-point errors at once:
  
  - ignore: Take no action when the exception occurs.
  - warn: Print a `RuntimeWarning` (via the Python `warnings` module).
  - raise: Raise a `FloatingPointError`.
  - call: Call a function specified using the `seterrcall` function.
  - print: Print a warning directly to `stdout`.
  - log: Record error in a `Log` object specified by `seterrcall`.

  The default is not to change the current behavior.

- `divide`: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  
  Treatment for division by zero.

- `over`: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  
  Treatment for floating-point overflow.

- `under`: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  
  Treatment for floating-point underflow.

- `invalid`: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  
  Treatment for invalid floating-point operation.

**Returns**

- `old_settings`: dict

  Dictionary containing the old settings.

**See Also:**

- `seterrcall`
  
  Set a callback function for the `call` mode.

- `geterr`, `geterrcall`

**Notes**

The floating-point exceptions are defined in the IEEE 754 standard [1]:

- Division by zero: infinite result obtained from finite numbers.
- Overflow: result too large to be expressed.
- Underflow: result so close to zero that some precision was lost.
- Invalid operation: result is not an expressible number, typically indicates that a NaN was produced.

**Examples**
>>> np.seterr(over='raise')
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}
>>> np.seterr(all='ignore')  # reset to default
{'over': 'raise', 'divide': 'warn', 'invalid': 'warn', 'under': 'warn'}

>>> np.int16(32000) * np.int16(3)
30464
>>> old_settings = np.seterr(all='warn', over='raise')
>>> np.int16(32000) * np.int16(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: overflow encountered in short_scalars

>>> np.seterr(all='print')
{'over': 'print', 'divide': 'print', 'invalid': 'print', 'under': 'print'}
>>> np.int16(32000) * np.int16(3)
Warning: overflow encountered in short_scalars
30464

Calling `seterr` with no arguments resets treatment for all floating-point errors to the defaults.

>>> old_settings = np.seterr()
>>> np.geterr()
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}

`seterrcall(func)`

Set the floating-point error callback function or log object.

There are two ways to capture floating-point error messages. The first is to set the error-handler to ‘call’, using `seterr`. Then, set the function to call using this function.

The second is to set the error-handler to ‘log’, using `seterr`. Floating-point errors then trigger a call to the ‘write’ method of the provided object.

Parameters

- `func`: callable f(err, flag) or object with write method

  Function to call upon floating-point errors (‘call’-mode) or object whose ‘write’ method is used to log such message (‘log’-mode).

  The call function takes two arguments. The first is the type of error (one of “divide”, “over”, “under”, or “invalid”), and the second is the status flag. The flag is a byte, whose least-significant bits indicate the status:

  \[ 0 \ 0 \ 0 \ 0 \ \text{invalid} \ \text{over} \ \text{under} \ \text{invalid} \]

  In other words, \( \text{flags} = \text{divide} + 2 \times \text{over} + 4 \times \text{under} + 8 \times \text{invalid} \).

  If an object is provided, its write method should take one argument, a string.

Returns

- `h`: callable, log instance or None

  The old error handler.
See Also:

seterr, geterr, geterrcall

Examples

Callback upon error:

```python
>>> def err_handler(type, flag):
    print "Floating point error (%s), with flag %s" % (type, flag)
...

>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')

>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([ Inf,  Inf,  Inf])

>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)

Log error message:

```python
>>> class Log(object):
    def write(self, msg):
        print "LOG: %s" % msg
...

>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')

>>> np.array([1, 2, 3]) / 0.0
LOG: Warning: divide by zero encountered in divide

>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)
```

2.5 Casting Rules

At the core of every ufunc is a one-dimensional strided loop that implements the actual function for a specific type combination. When a ufunc is created, it is given a static list of inner loops and a corresponding list of type signatures over which the ufunc operates. The ufunc machinery uses this list to determine which inner loop to use for a particular case. You can inspect the .types attribute for a particular ufunc to see which type combinations have a defined inner loop and which output type they produce (character codes are used in said output for brevity).

Casting must be done on one or more of the inputs whenever the ufunc does not have a core loop implementation for the input types provided. If an implementation for the input types cannot be found, then the algorithm searches for an implementation with a type signature to which all of the inputs can be cast “safely.” The first one it finds in its internal list of loops is selected and performed, after all necessary type casting. Recall that internal copies during ufuncs (even for casting) are limited to the size of an internal buffer (which is user settable).
Note: Universal functions in NumPy are flexible enough to have mixed type signatures. Thus, for example, a universal function could be defined that works with floating-point and integer values. See `ldexp` for an example.

By the above description, the casting rules are essentially implemented by the question of when a data type can be cast “safely” to another data type. The answer to this question can be determined in Python with a function call: `can_cast(fromtype, totype)`. The Figure below shows the results of this call for the 21 internally supported types on the author’s 32-bit system. You can generate this table for your system with the code given in the Figure.

**Figure**

Code segment showing the “can cast safely” table for a 32-bit system.

```python
>>> def print_table(ntypes):
...     print 'X',
...     for char in ntypes: print char,
...     print
...     for row in ntypes:
...         print row,
...         for col in ntypes:
...             print int(np.can_cast(row, col)),
...         print

>>> print_table(np.typecodes['All'])
X ? b h i l q p B H I L Q P f d g F D G S U V O
? 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
b 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
h 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
i 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
l 1 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0 1 1 1 1 1
q 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1
p 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1
B 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
H 0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
I 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1
L 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1
Q 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 1 1 1 1
P 0 0 0 0 0 0 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1
f 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
U 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
V 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
O 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

You should note that, while included in the table for completeness, the ‘S’, ‘U’, and ‘V’ types cannot be operated on by ufuncs. Also, note that on a 64-bit system the integer types may have different sizes, resulting in a slightly altered table.

Mixed scalar-array operations use a different set of casting rules that ensure that a scalar cannot “upcast” an array unless the scalar is of a fundamentally different kind of data (i.e., under a different hierarchy in the data-type hierarchy) than the array. This rule enables you to use scalar constants in your code (which, as Python types, are interpreted accordingly in ufuncs) without worrying about whether the precision of the scalar constant will cause upcasting on your large (small precision) array.
2.6 ufunc

2.6.1 Optional keyword arguments

All ufuncs take optional keyword arguments. These represent rather advanced usage and will not typically be used by most Numpy users. **sig**

Either a data-type, a tuple of data-types, or a special signature string indicating the input and output types of a ufunc. This argument allows you to provide a specific signature for the 1-d loop to use in the underlying calculation. If the loop specified does not exist for the ufunc, then a TypeError is raised. Normally, a suitable loop is found automatically by comparing the input types with what is available and searching for a loop with data-types to which all inputs can be cast safely. This keyword argument lets you bypass that search and choose a particular loop. A list of available signatures is provided by the **types** attribute of the ufunc object.

**extobj**

A list of length 1, 2, or 3 specifying the ufunc buffer-size, the error mode integer, and the error callback function. Normally, these values are looked up in a thread-specific dictionary. Passing them here circumvents that look up and uses the low-level specification provided for the error mode. This may be useful, for example, as an optimization for calculations requiring many ufunc calls on small arrays in a loop.

2.6.2 Attributes

There are some informational attributes that universal functions possess. None of the attributes can be set.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>doc</strong></td>
<td>A docstring for each ufunc. The first part of the docstring is dynamically generated from the number of outputs, the name, and the number of inputs. The second part of the docstring is provided at creation time and stored with the ufunc.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The name of the ufunc.</td>
</tr>
<tr>
<td>ufunc.nin</td>
<td>The number of inputs.</td>
</tr>
<tr>
<td>ufunc.nout</td>
<td>The number of outputs.</td>
</tr>
<tr>
<td>ufunc.nargs</td>
<td>The number of arguments.</td>
</tr>
<tr>
<td>ufunc.ntypes</td>
<td>The number of types.</td>
</tr>
<tr>
<td>ufunc.types</td>
<td>Returns a list with types grouped input-&gt;output.</td>
</tr>
<tr>
<td>ufunc.identity</td>
<td>The identity value.</td>
</tr>
</tbody>
</table>

**nin**

The number of inputs.

Data attribute containing the number of arguments the ufunc treats as input.

**Examples**

```python
>>> np.add.nin
2
>>> np.multiply.nin
2
>>> np.power.nin
2
>>> np.exp.nin
1
```

**nout**

The number of outputs.
Data attribute containing the number of arguments the ufunc treats as output.

**Notes**

Since all ufuncs can take output arguments, this will always be (at least) 1.

**Examples**

```python
>>> np.add.nout
1
>>> np.multiply.nout
1
>>> np.power.nout
1
>>> np.exp.nout
1
```

**nargs**

The number of arguments.

Data attribute containing the number of arguments the ufunc takes, including optional ones.

**Notes**

Typically this value will be one more than what you might expect because all ufuncs take the optional “out” argument.

**Examples**

```python
>>> np.add.nargs
3
>>> np.multiply.nargs
3
>>> np.power.nargs
3
>>> np.exp.nargs
2
```

**ntypes**

The number of types.

The number of numerical NumPy types - of which there are 18 total - on which the ufunc can operate.

**See Also:**

`numpy.ufunc.types`

**Examples**

```python
>>> np.add.ntypes
18
>>> np.multiply.ntypes
18
>>> np.power.ntypes
17
>>> np.exp.ntypes
7
>>> np.remainder.ntypes
14
```
types

Returns a list with types grouped input->output.

Data attribute listing the data-type “Domain-Range” groupings the ufunc can deliver. The data-types are given using the character codes.

See Also:

numpy.ufunc.ntypes

Examples

```python
>>> np.add.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
 'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
 'GG->G', 'OO->O']

>>> np.multiply.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
 'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
 'GG->G', 'OO->O']

>>> np.power.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
 'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
 'GG->G', 'OO->O']

>>> np.exp.types
['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']

>>> np.remainder.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
 'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']
```

identity

The identity value.

Data attribute containing the identity element for the ufunc, if it has one. If it does not, the attribute value is None.

Examples

```python
>>> np.add.identity
0

>>> np.multiply.identity
1

>>> np.power.identity
1

>>> print np.exp.identity
None
```

2.6.3 Methods

All ufuncs have four methods. However, these methods only make sense on ufuncs that take two input arguments and return one output argument. Attempting to call these methods on other ufuncs will cause a ValueError. The
reduce-like methods all take an `axis` keyword and a `dtype` keyword, and the arrays must all have dimension $\geq 1$. The `axis` keyword specifies the axis of the array over which the reduction will take place and may be negative, but must be an integer. The `dtype` keyword allows you to manage a very common problem that arises when naively using `{op}.reduce`. Sometimes you may have an array of a certain data type and wish to add up all of its elements, but the result does not fit into the data type of the array. This commonly happens if you have an array of single-byte integers. The `dtype` keyword allows you to alter the data type over which the reduction takes place (and therefore the type of the output). Thus, you can ensure that the output is a data type with precision large enough to handle your output. The responsibility of altering the reduce type is mostly up to you. There is one exception: if no `dtype` is given for a reduction on the “add” or “multiply” operations, then if the input type is an integer (or Boolean) data-type and smaller than the size of the `int_` data type, it will be internally upcast to the `int_` (or `uint`) data-type.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ufunc.reduce(a[, axis, dtype, out])</code></td>
<td>Reduces <code>a</code>'s dimension by one, by applying <code>ufunc</code> along one axis.</td>
</tr>
<tr>
<td><code>ufunc.accumulate(array[, axis, dtype, out])</code></td>
<td>Accumulate the result of applying the operator to all elements.</td>
</tr>
<tr>
<td><code>ufunc.reduceat(a, indices[, axis, dtype, out])</code></td>
<td>Performs a (local) reduce with specified slices over a single axis.</td>
</tr>
<tr>
<td><code>ufunc.outer(A, B)</code></td>
<td>Apply the <code>ufunc</code> <code>op</code> to all pairs (a, b) with a in A and b in B.</td>
</tr>
</tbody>
</table>

reduce

Reduces `a`’s dimension by one, by applying `ufunc` along one axis.

Let $a.shape = (N_0, ..., N_i, ..., N_{M-1})$. Then $ufunc.reduce(a, axis = i)[k_0, ..., k_{i-1}, k_{i+1}, ..., k_{M-1}] = \text{the result of iterating } j \text{ over } range(N_i), \text{ cumulatively applying } ufunc \text{ to each } a[k_0, ..., k_{i-1}, j, k_{i+1}, ..., k_{M-1}]$. For a one-dimensional array, reduce produces results equivalent to:

```python
r = op.identity # op = ufunc
for i in xrange(len(a)):
    r = op(r, a[i])
return r
```

For example, `add.reduce()` is equivalent to `sum()`.

**Parameters**

- `a`: array_like
  - The array to act on.
- `axis`: int, optional
  - The axis along which to apply the reduction.
- `dtype`: data-type code, optional
  - The type used to represent the intermediate results. Defaults to the data-type of the output array if this is provided, or the data-type of the input array if no output array is provided.
- `out`: ndarray, optional
  - A location into which the result is stored. If not provided, a freshly-allocated array is returned.

**Returns**

- `r`: ndarray
  - The reduced array. If `out` was supplied, `r` is a reference to it.

**Examples**

```python
>>> np.multiply.reduce([2, 3, 5])
30
```

A multi-dimensional array example:
accumulate()  
Accumulate the result of applying the operator to all elements.

For a one-dimensional array, accumulate produces results equivalent to:

```python
r = np.empty(len(A))
t = op.identity  # op = the ufunc being applied to A's elements
for i in xrange(len(A)):
    t = op(t, A[i])
    r[i] = t
return r
```

For example, add.accumulate() is equivalent to np.cumsum().

For a multi-dimensional array, accumulate is applied along only one axis (axis zero by default; see Examples below) so repeated use is necessary if one wants to accumulate over multiple axes.

**Parameters**

- `array` : array_like
  The array to act on.

- `axis` : int, optional
  The axis along which to apply the accumulation; default is zero.

- `dtype` : data-type code, optional
  The data-type used to represent the intermediate results. Defaults to the data-type of the output array if such is provided, or the the data-type of the input array if no output array is provided.

- `out` : ndarray, optional
  A location into which the result is stored. If not provided a freshly-allocated array is returned.

**Returns**

- `r` : ndarray
  The accumulated values. If `out` was supplied, `r` is a reference to `out`.  

```python
>>> X = np.arange(8).reshape((2,2,2))
>>> X
array([[[0, 1],
        [2, 3]],
        [[4, 5],
        [6, 7]]])
>>> np.add.reduce(X, 0)
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X)  # confirm: default axis value is 0
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X, 1)
array([[ 2,  4],
       [10, 12]])
>>> np.add.reduce(X, 2)
array([[ 1,  5],
       [ 9, 13]])
```
Examples

1-D array examples:

```python
>>> np.add.accumulate([2, 3, 5])
array([ 2,  5, 10])
```

```python
>>> np.multiply.accumulate([2, 3, 5])
array([ 2,  6, 30])
```

2-D array examples:

```python
>>> I = np.eye(2)
```

```python
>>> I
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Accumulate along axis 0 (rows), down columns:

```python
>>> np.add.accumulate(I, 0)
array([[ 1.,  0.],
       [ 1.,  1.]])
```

```python
>>> np.add.accumulate(I)  # no axis specified = axis zero
array([[ 1.,  0.],
       [ 1.,  1.]])
```

Accumulate along axis 1 (columns), through rows:

```python
>>> np.add.accumulate(I, 1)
array([[ 1.,  1.],
       [ 0.,  1.]])
```

redecat()

Performs a (local) reduce with specified slices over a single axis.

For i in range(len(indices)), redecat computes ufunc.reduce(a[indices[i]:indices[i+1]]), which becomes the i-th generalized “row” parallel to axis in the final result (i.e., in a 2-D array, for example, if axis = 0, it becomes the i-th row, but if axis = 1, it becomes the i-th column). There are two exceptions to this:

- when i = len(indices) - 1 (so for the last index), indices[i+1] = a.shape[axis].
- if indices[i] >= indices[i + 1], the i-th generalized “row” is simply a[indices[i]].

The shape of the output depends on the size of indices, and may be larger than a (this happens if len(indices) > a.shape[axis]).

Parameters

- a : array_like
  The array to act on.

- indices : array_like
  Paired indices, comma separated (not colon), specifying slices to reduce.

- axis : int, optional
  The axis along which to apply the redecat.

- dtype : data-type code, optional
The type used to represent the intermediate results. Defaults to the data type of the output array if this is provided, or the data type of the input array if no output array is provided.

**out**: ndarray, optional

A location into which the result is stored. If not provided a freshly-allocated array is returned.

**Returns**

* r : ndarray

The reduced values. If *out* was supplied, *r* is a reference to *out*.

**Notes**

A descriptive example:

If *a* is 1-D, the function `ufunc.accumulate(*a*)` is the same as `ufunc.reduceat(*a*, indices)[::2]` where `indices` is `range(len(*array*) - 1)` with a zero placed in every other element: `indices = zeros(2 * len(*a*) - 1), indices[1::2] = range(1, len(*a*)).`

Don’t be fooled by this attribute’s name: `reduceat(*a*)` is not necessarily smaller than *a*.

**Examples**

To take the running sum of four successive values:

```python
>>> np.add.reduceat(np.arange(8), [0, 4, 1, 5, 2, 6, 3, 7])[::2]
array([ 6, 10, 14, 18])
```

A 2-D example:

```python
>>> x = np.linspace(0, 15, 16).reshape(4, 4)
>>> x
array([[ 0., 1., 2., 3.],
       [ 4., 5., 6., 7.],
       [ 8., 9., 10., 11.],
       [12., 13., 14., 15.]])

# reduce such that the result has the following five rows:
# [row1 + row2 + row3]
# [row4]
# [row2]
# [row3]
# [row1 + row2 + row3 + row4]

>>> np.add.reduceat(x, [0, 3, 1, 2, 0])
array([[12., 15., 18., 21.],
       [12., 13., 14., 15.],
       [ 4.,  5.,  6.,  7.],
       [ 0.,  9., 10., 11.],
       [24., 28., 32., 36.]])

# reduce such that result has the following two columns:
# [col1 * col2 * col3, col4]
```
>>> np.multiply.reduceat(x, [0, 3], 1)
array([[ 0.,  3.],
       [120.,  7.],
       [ 720., 11.],
       [2184., 15.]]

outer()

Apply the ufunc op to all pairs (a, b) with a in A and b in B.
Let M = A.ndim, N = B.ndim. Then the result, C, of op.outer(A, B) is an array of dimension M + N such that:

\[ C[i_0, \ldots, i_{M-1}, j_0, \ldots, j_{N-1}] = op(A[i_0, \ldots, i_{M-1}], B[j_0, \ldots, j_{N-1}]) \]

For A and B one-dimensional, this is equivalent to:

```python
r = empty(len(A), len(B))
for i in xrange(len(A)):
    for j in xrange(len(B)):
        r[i, j] = op(A[i], B[j]) # op = ufunc in question
```

Parameters

- **A**: array_like
  - First array

- **B**: array_like
  - Second array

Returns

- **r**: ndarray
  - Output array

See Also:

- `numpy.outer`

Examples

```python
>>> np.multiply.outer([1, 2, 3], [4, 5, 6])
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])
```

A multi-dimensional example:

```python
>>> A = np.array([[[1, 2, 3], [4, 5, 6]]])
>>> A.shape
(2, 3)
>>> B = np.array([[[1, 2, 3], [4]]])
>>> B.shape
(1, 4)
>>> C = np.multiply.outer(A, B)
>>> C.shape; C
(2, 3, 1, 4)
```
array([[[ 1,  2,  3,  4]],
       [[ 2,  4,  6,  8]],
       [[ 3,  6,  9, 12]]],
       [[[ 4,  8, 12, 16]],
       [[ 5, 10, 15, 20]],
       [[ 6, 12, 18, 24]]])

**Warning:** A reduce-like operation on an array with a data-type that has a range “too small” to handle the result will silently wrap. One should use `dtype` to increase the size of the data-type over which reduction takes place.

### 2.7 Available ufuncs

There are currently more than 60 universal functions defined in `numpy` on one or more types, covering a wide variety of operations. Some of these ufuncs are called automatically on arrays when the relevant infix notation is used (e.g., `add(a, b)` is called internally when `a + b` is written and `a or b` is an `ndarray`). Nevertheless, you may still want to use the ufunc call in order to use the optional output argument(s) to place the output(s) in an object (or objects) of your choice.

Recall that each ufunc operates element-by-element. Therefore, each ufunc will be described as if acting on a set of scalar inputs to return a set of scalar outputs.

**Note:** The ufunc still returns its output(s) even if you use the optional output argument(s).
### 2.7.1 Math operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x1, x2, out)</td>
<td>Add arguments element-wise.</td>
</tr>
<tr>
<td>subtract(x1, x2, out)</td>
<td>Subtract arguments, element-wise.</td>
</tr>
<tr>
<td>multiply(x1, x2, out)</td>
<td>Multiply arguments element-wise.</td>
</tr>
<tr>
<td>divide(x1, x2, out)</td>
<td>Divide arguments element-wise.</td>
</tr>
<tr>
<td>logaddexp(x1, x2, out)</td>
<td>Logarithm of the sum of exponentiations of the inputs.</td>
</tr>
<tr>
<td>logaddexp2(x1, x2, out)</td>
<td>Logarithm of the sum of exponentiations of the inputs in base-2.</td>
</tr>
<tr>
<td>true_divide(x1, x2, out)</td>
<td>Returns a true division of the inputs, element-wise.</td>
</tr>
<tr>
<td>floor_divide(x1, x2, out)</td>
<td>Return the largest integer smaller or equal to the division of the inputs.</td>
</tr>
<tr>
<td>negative(x)</td>
<td>Return the negative of each element of the original array.</td>
</tr>
<tr>
<td>power(x1, x2, out)</td>
<td>Return the element-wise remainder of division.</td>
</tr>
<tr>
<td>mod(x1, x2, out)</td>
<td>Calculate the absolute value element-wise.</td>
</tr>
<tr>
<td>floor(x, out)</td>
<td>Round elements of the array to the nearest integer.</td>
</tr>
<tr>
<td>sign(x, out)</td>
<td>Return the element-wise indication of the sign of a number.</td>
</tr>
<tr>
<td>conj(x, out)</td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td>exp(x, out)</td>
<td>Calculate the exponential of all elements in the input array.</td>
</tr>
<tr>
<td>exp2(x, out)</td>
<td>Calculate (2^p) for all (p) in the input array.</td>
</tr>
<tr>
<td>log(x, out)</td>
<td>Natural logarithm, element-wise.</td>
</tr>
<tr>
<td>log2(x, y)</td>
<td>Return the base 2 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td>log10(x, out)</td>
<td>Return the base 10 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td>expm1(x, out)</td>
<td>Calculate (\exp(x) - 1) for all elements in the array.</td>
</tr>
<tr>
<td>log1p(x, out)</td>
<td>Return the natural logarithm of one plus the input array, element-wise.</td>
</tr>
<tr>
<td>sqrt(x, out)</td>
<td>Return the positive square-root of an array, element-wise.</td>
</tr>
<tr>
<td>square(x, out)</td>
<td>Return the element-wise square of the input.</td>
</tr>
<tr>
<td>reciprocal(x, out)</td>
<td>Return the reciprocal of the argument, element-wise.</td>
</tr>
<tr>
<td>ones_like(x, out)</td>
<td>Returns an array of ones with the same shape and type as a given array.</td>
</tr>
</tbody>
</table>

**Tip:** The optional output arguments can be used to help you save memory for large calculations. If your arrays are large, complicated expressions can take longer than absolutely necessary due to the creation and (later) destruction of temporary calculation spaces. For example, the expression \(G = a * b + c\) is equivalent to \(t1 = A * B; G = T1 + C; \) del \(t1\). It will be more quickly executed as \(G = A * B; \) add(G, C, G) which is the same as \(G = A * B; G += C\).

### 2.7.2 Trigonometric functions

All trigonometric functions use radians when an angle is called for. The ratio of degrees to radians is \(180^\circ / \pi\).
### 2.7.3 Bit-twiddling functions

These functions all require integer arguments and they manipulate the bit-pattern of those arguments.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bitwise_and (x1, x2[, out])</code></td>
<td>Compute the bit-wise AND of two arrays element-wise.</td>
</tr>
<tr>
<td><code>bitwise_or (x1, x2[, out])</code></td>
<td>Compute the bit-wise OR of two arrays element-wise.</td>
</tr>
<tr>
<td><code>bitwise_xor (x1, x2[, out])</code></td>
<td>Compute the bit-wise XOR of two arrays element-wise.</td>
</tr>
<tr>
<td><code>invert (x[, out])</code></td>
<td>Compute bit-wise inversion, or bit-wise NOT, element-wise.</td>
</tr>
<tr>
<td><code>left_shift (x1, x2[, out])</code></td>
<td>Shift the bits of an integer to the left.</td>
</tr>
<tr>
<td><code>right_shift (x1, x2[, out])</code></td>
<td>Shift the bits of an integer to the right.</td>
</tr>
</tbody>
</table>

### 2.7.4 Comparison functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>greater (x1, x2[, out])</code></td>
<td>Return (x1 &gt; x2) element-wise.</td>
</tr>
<tr>
<td><code>greater_equal (x1, x2[, out])</code></td>
<td>Return (x1 \geq x2) element-wise.</td>
</tr>
<tr>
<td><code>less (x1, x2[, out])</code></td>
<td>Return (x1 &lt; x2) element-wise.</td>
</tr>
<tr>
<td><code>less_equal (x1, x2[, out])</code></td>
<td>Return (x1 \leq x2) element-wise.</td>
</tr>
<tr>
<td><code>not_equal (x1, x2[, out])</code></td>
<td>Return (x1 \neq x2) element-wise.</td>
</tr>
<tr>
<td><code>equal (x1, x2[, out])</code></td>
<td>Return (x1 = x2) element-wise.</td>
</tr>
</tbody>
</table>

**Warning:** Do not use the Python keywords `and` and `or` to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators `&` and `|` instead.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>logical_and (x1, x2[, out])</code></td>
<td>Compute the truth value of (x1 &amp; x2) elementwise.</td>
</tr>
<tr>
<td><code>logical_or (x1, x2[, out])</code></td>
<td>Compute the truth value of (x1 | x2) elementwise.</td>
</tr>
<tr>
<td><code>logical_xor (x1, x2[, out])</code></td>
<td>Compute the truth value of (x1 \oplus x2) elementwise.</td>
</tr>
<tr>
<td><code>logical_not (x[, out])</code></td>
<td>Compute the truth value of (\neg x) elementwise.</td>
</tr>
</tbody>
</table>

**Warning:** The bit-wise operators `&` and `|` are the proper way to perform element-by-element array comparisons. Be sure you understand the operator precedence: \((a > 2) & (a < 5)\) is the proper syntax because \(a > 2\) & \(a < 5\) will result in an error due to the fact that \(2 & a\) is evaluated first.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maximum (x1, x2[, out])</code></td>
<td>Element-wise maximum of array elements.</td>
</tr>
</tbody>
</table>
Tip: The Python function `max()` will find the maximum over a one-dimensional array, but it will do so using a slower sequence interface. The reduce method of the maximum ufunc is much faster. Also, the `max()` method will not give answers you might expect for arrays with greater than one dimension. The reduce method of minimum also allows you to compute a total minimum over an array.

```
minimum(x1, x2[, out])  Element-wise minimum of array elements.
```

**Warning:** the behavior of `maximum(a, b)` is different than that of `max(a, b)`. As a ufunc, `maximum(a, b)` performs an element-by-element comparison of `a` and `b` and chooses each element of the result according to which element in the two arrays is larger. In contrast, `max(a, b)` treats the objects `a` and `b` as a whole, looks at the (total) truth value of `a > b` and uses it to return either `a` or `b` (as a whole). A similar difference exists between `minimum(a, b)` and `min(a, b)`.

### 2.7.5 Floating functions

Recall that all of these functions work element-by-element over an array, returning an array output. The description details only a single operation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isreal(x)</code></td>
<td>Returns a bool array, where True if input element is real.</td>
</tr>
<tr>
<td><code>iscomplex(x)</code></td>
<td>Returns a bool array, where True if input element is complex.</td>
</tr>
<tr>
<td><code>isfinite(x[, out])</code></td>
<td>Test element-wise for finite-ness (not infinity or not Not a Number).</td>
</tr>
<tr>
<td><code>isnan(x[, out])</code></td>
<td>Test element-wise for Not a Number (NaN), return result as a bool array.</td>
</tr>
<tr>
<td><code>copysign(x1, x2[, out])</code></td>
<td>Change the sign of <code>x1</code> to that of <code>x2</code>, element-wise.</td>
</tr>
<tr>
<td><code>nextafter(x1, x2[, out])</code></td>
<td>Return the next representable floating-point value after <code>x1</code> in the direction of <code>x2</code>.</td>
</tr>
<tr>
<td><code>modf(x[, out1, out2])</code></td>
<td>Return the fractional and integral parts of an array, element-wise.</td>
</tr>
<tr>
<td><code>ldexp(x1, x2[, out])</code></td>
<td>Compute <code>y = x1 * 2**x2</code>.</td>
</tr>
<tr>
<td><code>frexp(x[, out1, out2])</code></td>
<td>Split the number, <code>x</code>, into a normalized fraction (<code>y1</code>) and exponent (<code>y2</code>).</td>
</tr>
<tr>
<td><code>floor(x[, out])</code></td>
<td>Return the floor of the input, element-wise.</td>
</tr>
<tr>
<td><code>ceil(x[, out])</code></td>
<td>Return the ceiling of the input, element-wise.</td>
</tr>
<tr>
<td><code>trunc(x[, out])</code></td>
<td>Return the truncated value of the input, element-wise.</td>
</tr>
</tbody>
</table>
3.1 Array creation routines

See Also:
Array creation (in NumPy User Guide)

3.1.1 Ones and zeros

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>empty(shape[, dtype, order])</code></td>
<td>Return a new array of given shape and type, without initializing entries.</td>
</tr>
<tr>
<td><code>empty_like(a)</code></td>
<td>Return a new array with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>eye(N[, M, k, dtype])</code></td>
<td>Return a 2-D array with ones on the diagonal and zeros elsewhere.</td>
</tr>
<tr>
<td><code>identity(n[, dtype])</code></td>
<td>Return the identity array.</td>
</tr>
<tr>
<td><code>ones(shape[, dtype, order])</code></td>
<td>Return a new array of given shape and type, filled with ones.</td>
</tr>
<tr>
<td><code>ones_like(x[, out])</code></td>
<td>Returns an array of ones with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>zeros(shape[, dtype, order])</code></td>
<td>Return a new array of given shape and type, filled with zeros.</td>
</tr>
<tr>
<td><code>zeros_like(a)</code></td>
<td>Return an array of zeros with the same shape and type as a given array.</td>
</tr>
</tbody>
</table>

**empty()**
Return a new array of given shape and type, without initializing entries.

**Parameters**

- `shape`: int or tuple of int
  Shape of the empty array
- `dtype`: data-type, optional
  Desired output data-type.
- `order`: {'C', 'F'}, optional
  Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

**See Also:**
`empty_like`, `zeros`, `ones`

**Notes**

`empty`, unlike `zeros`, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.
Examples

```python
>>> np.empty([2, 2])
array([[ -9.74499359e+001, 6.69583040e-309],
       [ 2.13182611e-314, 3.06959433e-309]])

>>> np.empty([2, 2], dtype=int)
array([[ -1073741821, -1067949133],
       [ 496041986, 19249760]])
```

e**mpty_like**(a)

Return a new array with the same shape and type as a given array.

**Parameters**

- **a**: array_like
  
The shape and data-type of *a* define the parameters of the returned array.

**Returns**

- **out**: ndarray
  
  Array of random data with the same shape and type as *a*.

See Also:

- **ones_like**: Return an array of ones with shape and type of input.
- **zeros_like**: Return an array of zeros with shape and type of input.
- **empty**: Return a new uninitialized array.
- **ones**: Return a new array setting values to one.
- **zeros**: Return a new array setting values to zero.

Notes

This function does *not* initialize the returned array; to do that use zeros_like or ones_like instead. It may be marginally faster than the functions that do set the array values.

Examples

```python
>>> a = ([1, 2, 3], [4, 5, 6])
# a is array-like
>>> np.empty_like(a)
array([[ -1073741821, -1073741821, 3],
       [ 0, 0, -1073741821]])

>>> a = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> np.empty_like(a)
array([[ -2.00000715e+000, 1.48219694e-323, -2.00000572e+000],
       [ 4.38791518e-305, -2.00000715e+000, 4.17269252e-309]])
```

e**ye** *(N, M=None, k=0, dtype='float')*

Return a 2-D array with ones on the diagonal and zeros elsewhere.

```python
```

288 Chapter 3. Routines
Parameters

N : int
   Number of rows in the output.

M : int, optional
   Number of columns in the output. If None, defaults to N.

k : int, optional
   Index of the diagonal: 0 refers to the main diagonal, a positive value refers to an upper
diagonal, and a negative value to a lower diagonal.

dtype : dtype, optional
   Data-type of the returned array.

Returns

I : ndarray (N,M)
   An array where all elements are equal to zero, except for the k-th diagonal, whose values
   are equal to one.

See Also:

diag
   Return a diagonal 2-D array using a 1-D array specified by the user.

Examples

>>> np.eye(2, dtype=int)
array([[1, 0],
       [0, 1]])

>>> np.eye(3, k=1)
array([[ 0., 1., 0.],
       [ 0., 0., 1.],
       [ 0., 0., 0.]])

identity (n, dtype=None)
   Return the identity array.

   The identity array is a square array with ones on the main diagonal.

Parameters

n : int
   Number of rows (and columns) in n x n output.

dtype : data-type, optional
   Data-type of the output. Defaults to float.

Returns

out : ndarray
   n x n array with its main diagonal set to one, and all other elements 0.

Examples

>>> np.identity(3)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
ones (shape, dtype=None, order=’C’)
Return a new array of given shape and type, filled with ones.

Please refer to the documentation for zeros.

See Also:
zeros

Examples

>>> np.ones(5)
array([1., 1., 1., 1., 1.])

>>> np.ones((5,), dtype=np.int)
array([1, 1, 1, 1, 1])

>>> np.ones((2, 1))
array([[1.],
       [1.]])

>>> s = (2,2)
>>> np.ones(s)
array([[1., 1.],
       [1., 1.]])

ones_like()
Returns an array of ones with the same shape and type as a given array.

Equivalent to a.copy().fill(1).

Please refer to the documentation for zeros_like.

See Also:
zeros_like

Examples

>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.ones_like(a)
array([[1, 1, 1],
       [1, 1, 1]])

zeros()
Return a new array of given shape and type, filled with zeros.

Parameters

shape : int or sequence of ints
Shape of the new array, e.g., (2, 3) or 2.

dtype : data-type, optional
The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.

order : {'C', 'F'}, optional
Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

- **out**: ndarray
  
  Array of zeros with the given shape, dtype, and order.

**See Also:**

- **zeros_like**: Return an array of zeros with shape and type of input.
- **ones_like**: Return an array of ones with shape and type of input.
- **empty_like**: Return an empty array with shape and type of input.
- **ones**: Return a new array setting values to one.
- **empty**: Return a new uninitialized array.

**Examples**

```python
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])

>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])

>>> s = (2,2)
>>> np.zeros(s)
array([[0., 0.],
       [0., 0.]])

>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

**zeros_like(a)**

Return an array of zeros with the same shape and type as a given array.

Equivalent to `a.copy().fill(0)`.

**Parameters**

- **a**: array_like

  The shape and data-type of `a` define the parameters of the returned array.

**Returns**

- **out**: ndarray
Array of zeros with same shape and type as a.

See Also:

ones_like
Return an array of ones with shape and type of input.

empty_like
Return an empty array with shape and type of input.

zeros
Return a new array setting values to zero.

ones
Return a new array setting values to one.

empty
Return a new uninitialized array.

Examples

```python
>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])

>>> np.zeros_like(x)
array([[0., 0., 0.],
       [0., 0., 0.]])

>>> y = np.arange(3, dtype=np.float)
>>> y
array([ 0., 1., 2.])

>>> np.zeros_like(y)
array([ 0., 0., 0.])
```

3.1.2 From existing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array()</td>
<td>Create an array.</td>
</tr>
<tr>
<td>asarray(a[, dtype, order])</td>
<td>Convert the input to an array.</td>
</tr>
<tr>
<td>asanyarray(a[, dtype, order])</td>
<td>Convert the input to a ndarray, but pass ndarray subclasses through.</td>
</tr>
<tr>
<td>ascontiguousarray(a[, dtype])</td>
<td>Return a contiguous array in memory (C order).</td>
</tr>
<tr>
<td>asmatrix(data[, dtype])</td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td>copy(a)</td>
<td>Return an array copy of the given object.</td>
</tr>
<tr>
<td>frombuffer(buffer[, dtype, count, offset])</td>
<td>Interpret a buffer as a 1-dimensional array.</td>
</tr>
<tr>
<td>fromfile(file[, dtype, count, sep])</td>
<td>Construct an array from data in a text or binary file.</td>
</tr>
<tr>
<td>fromfunction(function, shape, **kwargs)</td>
<td>Construct an array by executing a function over each coordinate.</td>
</tr>
<tr>
<td>fromiter(iterable, dtype[, count])</td>
<td>Create a new 1-dimensional array from an iterable object.</td>
</tr>
<tr>
<td>fromstring(string[, dtype, count, sep])</td>
<td>Return a new 1-D array initialized from raw binary or text data in string.</td>
</tr>
<tr>
<td>.loadtxt(fname[, dtype, comments, ...])</td>
<td>Load data from a text file.</td>
</tr>
</tbody>
</table>
Parameters

object : array_like

An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype : data-type, optional

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to ‘upcast’ the array. For downcasting, use the .astype(t) method.

copy : bool, optional

If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (dtype, order, etc.).

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

Specify the order of the array. If order is ‘C’ (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is ‘F’, then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is ‘A’, then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

subok : bool, optional

If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

ndmin : int, optional

Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

Examples

>>> np.array([1, 2, 3])
array([1, 2, 3])

Upcasting:

>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])

More than one dimension:

>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])

Minimum dimensions 2:

>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])

Type provided:
```python
>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j, 2.+0.j, 3.+0.j])
```

Data-type consisting of more than one element:

```python
>>> x = np.array(((1,2),(3,4)),dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```python
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```python
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
       [3, 4]])
```

`asarray(a, dtype=None, order=None)`

Convert the input to an array.

**Parameters**

- `a`: array_like
  - Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples of tuples, tuples of lists and ndarrays.

- `dtype`: data-type, optional
  - By default, the data-type is inferred from the input data.

- `order`: {'C', 'F'}, optional
  - Whether to use row-major ('C') or column-major ('F' for FORTRAN) memory representation. Defaults to ‘C’.

**Returns**

- `out`: ndarray
  - Array interpretation of `a`. No copy is performed if the input is already an ndarray. If `a` is a subclass of ndarray, a base class ndarray is returned.

**See Also:**

- `asanyarray`
  - Similar function which passes through subclasses.

- `ascontiguousarray`
  - Convert input to a contiguous array.

- `asfarray`
  - Convert input to a floating point ndarray.

- `asfortranarray`
  - Convert input to an ndarray with column-major memory order.

- `asarray_chkfinite`
  - Similar function which checks input for NaNs and Infs.
fromiter
Create an array from an iterator.

fromfunction
Construct an array by executing a function on grid positions.

Examples
Convert a list into an array:

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

Existing arrays are not copied:

```python
>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True
```

Contrary to `asanyarray`, ndarray subclasses are not passed through:

```python
>>> issubclass(np.matrix, np.ndarray)
True
>>> a = np.matrix([[1, 2]])
>>> np.asarray(a) is a
False
>>> np.asanyarray(a) is a
True
```

`asanyarray (a, dtype=None, order=None)`
Convert the input to a ndarray, but pass ndarray subclasses through.

Parameters
- `a`: array_like
  Input data, in any form that can be converted to an array. This includes scalars, lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.
- `dtype`: data-type, optional
  By default, the data-type is inferred from the input data.
- `order`: {'C', 'F'}, optional
  Whether to use row-major (‘C’) or column-major (‘F’) memory representation. Defaults to ‘C’.

Returns
- `out`: ndarray or an ndarray subclass
  Array interpretation of a. If a is an ndarray or a subclass of ndarray, it is returned as-is and no copy is performed.

See Also:
- `asarray`
  Similar function which always returns ndarrays.
- `ascontiguousarray`
  Convert input to a contiguous array.

3.1. Array creation routines 295
**asfarray**
Convert input to a floating point ndarray.

**asfortranarray**
Convert input to an ndarray with column-major memory order.

**asarray_chkfinite**
Similar function which checks input for NaNs and Infs.

**fromiter**
Create an array from an iterator.

**fromfunction**
Construct an array by executing a function on grid positions.

### Examples
Convert a list into an array:

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

Instances of *ndarray* subclasses are passed through as-is:

```python
>>> a = np.matrix([1, 2])
>>> np.asanyarray(a) is a
True
```

**ascontiguousarray** *(a, dtype=None)*
Return a contiguous array in memory (C order).

**Parameters**
- **a** : array_like
  Input array.
- **dtype** : str or dtype object, optional
  Data-type of returned array.

**Returns**
- **out** : ndarray
  Contiguous array of same shape and content as *a*, with type *dtype* if specified.

**See Also:**

**asfortranarray**
Convert input to an ndarray with column-major memory order.

**require**
Return an ndarray that satisfies requirements.

**ndarray.flags**
Information about the memory layout of the array.

### Examples
>>> x = np.arange(6).reshape(2,3)
>>> np.ascontiguousarray(x, dtype=np.float32)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.flags['C_CONTIGUOUS']
True

asmatrix (data, dtype=None)
Interpret the input as a matrix.

Unlike matrix, asmatrix does not make a copy if the input is already a matrix or an ndarray. Equivalent to matrix(data, copy=False).

Parameters
data : array_like
Input data.

Returns
mat : matrix
data interpreted as a matrix.

Examples

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

>>> m = np.asmatrix(x)

>>> x[0,0] = 5

>>> m
matrix([[5, 2],
        [3, 4]])

copy (a)
Return an array copy of the given object.

Parameters
a : array_like
Input data.

Returns
arr : ndarray
Array interpretation of a.

Notes
This is equivalent to

>>> np.array(a, copy=True)
Examples

Create an array x, with a reference y and a copy z:

```python
>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)
```

Note that, when we modify x, y changes, but not z:

```python
>>> x[0] = 10
>>> x[0] == y[0]  # True
>>> x[0] == z[0]  # False
```

**frombuffer()**

Interpret a buffer as a 1-dimensional array.

**Parameters**

- `buffer`: An object that exposes the buffer interface.
- `dtype`: data-type, optional
  Data type of the returned array.
- `count`: int, optional
  Number of items to read. -1 means all data in the buffer.
- `offset`: int, optional
  Start reading the buffer from this offset.

**Notes**

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```python
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')  # Big-endian byte-order
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

**Examples**

```python
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

**fromfile**

Construct an array from data in a text or binary file.

A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the `tofile` method can be read using this function.

**Parameters**

- `file`: file or str
Open file object or filename.

dtype : data-type

Data type of the returned array. For binary files, it is used to determine the size and byte-order of the items in the file.

count : int

Number of items to read. -1 means all items (i.e., the complete file).

sep : str

Separator between items if file is a text file. Empty ("") separator means the file should be treated as binary. Spaces (" ") in the separator match zero or more whitespace characters. A separator consisting only of spaces must match at least one whitespace.

See Also:
load, save, ndarray.tofile

loadtxt

More flexible way of loading data from a text file.

Notes

Do not rely on the combination of tofile and fromfile for data storage, as the binary files generated are are not platform independent. In particular, no byte-order or data-type information is saved. Data can be stored in the platform independent .npy format using save and load instead.

Examples

Construct an ndarray:

```python
>>> dt = np.dtype([('time', [(‘min’, int), (‘sec’, int)]),
...   (‘temp’, float))
>>> x = np.zeros((1,), dtype=dt)
>>> x[‘time’][‘min’] = 10; x[‘temp’] = 98.25
>>> x
array([(10, 0), 98.25],
   dtype=[(‘time’, [(‘min’, ‘<i4’), (‘sec’, ‘<i4’)]), (‘temp’, ‘<f8’)])
```

Save the raw data to disk:

```python
>>> import os
>>> fname = os.tmpnam()
>>> x.tofile(fname)
```

Read the raw data from disk:

```python
>>> np.fromfile(fname, dtype=dt)
array([(10, 0), 98.25],
   dtype=[(‘time’, [(‘min’, ‘<i4’), (‘sec’, ‘<i4’)]), (‘temp’, ‘<f8’)])
```

The recommended way to store and load data:

```python
>>> np.save(fname, x)
>>> np.load(fname + ‘.npy’)
array([(10, 0), 98.25],
   dtype=[(‘time’, [(‘min’, ‘<i4’), (‘sec’, ‘<i4’)]), (‘temp’, ‘<f8’)])
```
fromfunction (function, shape, **kwargs)
Construct an array by executing a function over each coordinate.

The resulting array therefore has a value fn(x, y, z) at coordinate (x, y, z).

Parameters

function : callable
The function is called with N parameters, each of which represents the coordinates of
the array varying along a specific axis. For example, if shape were (2, 2), then the
parameters would be two arrays, [[0, 0], [1, 1]] and [[0, 1], [0, 1]].
function must be capable of operating on arrays, and should return a scalar value.

shape : (N,) tuple of ints
Shape of the output array, which also determines the shape of the coordinate arrays
passed to function.

dtype : data-type, optional
Data-type of the coordinate arrays passed to function. By default, dtype is float.

Returns

out : any
The result of the call to function is passed back directly. Therefore the type and shape
of out is completely determined by function.

See Also:
indices, meshgrid

Notes

Keywords other than shape and dtype are passed to function.

Examples

>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)

>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

fromiter()
Create a new 1-dimensional array from an iterable object.

Parameters

iterable : iterable object
An iterable object providing data for the array.

dtype : data-type
The data type of the returned array.

count : int, optional
The number of items to read from iterable. The default is -1, which means all data is read.

Returns

out : ndarray

The output array.

Notes

Specify count to improve performance. It allows fromiter to pre-allocate the output array, instead of resizing it on demand.

Examples

>>> iterable = (x*x for x in range(5))
>>> np.fromiter(iterable, np.float)
array([ 0., 1., 4., 9., 16.])

fromstring()

Return a new 1-D array initialized from raw binary or text data in string.

Parameters

string : str

A string containing the data.

dtype : dtype, optional

The data type of the array. For binary input data, the data must be in exactly this format.

count : int, optional

Read this number of dtype elements from the data. If this is negative, then the size will be determined from the length of the data.

sep : str, optional

If provided and not empty, then the data will be interpreted as ASCII text with decimal numbers. This argument is interpreted as the string separating numbers in the data. Extra whitespace between elements is also ignored.

Returns

arr : array

The constructed array.

Raises

ValueError :

If the string is not the correct size to satisfy the requested dtype and count.

Examples

>>> np.fromstring('x01x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
>>> np.fromstring('x01x02x03x04x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
Invalid inputs:

```python
>>> np.fromstring('\x01\x02\x03\x04\x05', dtype=np.int32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: string size must be a multiple of element size
```

```python
>>> np.fromstring('\x01\x02', dtype=np.uint8, count=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: string is smaller than requested size
```

`loadtxt`

```python
(loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False)
```

Load data from a text file.

Each row in the text file must have the same number of values.

**Parameters**

- `fname`: file or str
  - File or filename to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.
- `dtype`: dtype, optional
  - Data type of the resulting array. If this is a record data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.
- `comments`: str, optional
  - The character used to indicate the start of a comment.
- `delimiter`: str, optional
  - The string used to separate values. By default, this is any whitespace.
- `converters`: dict, optional
  - A dictionary mapping column number to a function that will convert that column to a float. E.g., if column 0 is a date string: `converters = {0: datestr2num}`. Converters can also be used to provide a default value for missing data: `converters = {3: lambda s: float(s or 0)}`.
- `skiprows`: int, optional
  - Skip the first `skiprows` lines.
- `usecols`: sequence, optional
  - Which columns to read, with 0 being the first. For example, `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns.
- `unpack`: bool, optional
  - If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`. Default is False.

**Returns**

- `out`: ndarray
  - Data read from the text file.
3.1.3 Creating record arrays (**numpy.rec**)

**Note:** `numpy.rec` is the preferred alias for `numpy.core.records`.

<table>
<thead>
<tr>
<th><code>core.records.array</code></th>
<th>Construct a record array from a wide-variety of objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(obj[, dtype, shape, offset,...])</code></td>
<td>create a record array from a (flat) list of arrays</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>core.records.fromarrays</code></th>
<th>create a recarray from a list of records in text form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(arrayList[, dtype, shape, formats,...])</code></td>
<td>create a (read-only) record array from binary data contained in a string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>core.records.fromstring</code></th>
<th>Create an array from binary file data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(datastring[, dtype, shape, offset,...])</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>core.records.fromfile</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(fd[, dtype, shape, offset,...])</code></td>
<td>Create an array from binary file data</td>
</tr>
</tbody>
</table>

---

**Examples**

```python
>>> from StringIO import StringIO  # StringIO behaves like a file object
>>> c = StringIO("0 1
2 3")
>>> np.loadtxt(c)
array([[ 0., 1.],
       [ 2., 3.]])

>>> d = StringIO("M 21 72
F 35 58")
>>> np.loadtxt(d, dtype=({'names': ('gender', 'age', 'weight'),
...                       'formats': ('S1', 'i4', 'f4')}))
array([('M', 21, 72), ('F', 35, 58)],
      dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])

>>> c = StringIO("1,0,2
3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([ 1., 3.])
>>> y
array([ 2., 4.])
```
```python
(2, 'dd', 2.0)
>>> x1[1]=34
>>> r.a
array([1, 2, 3, 4])

fromrecords (recList, dtype=None, shape=None, formats=None, names=None, titles=None, aligned=False, byteorder=None)
create a recarray from a list of records in text form

The data in the same field can be heterogeneous, they will be promoted to the highest data type. This
method is intended for creating smaller record arrays. If used to create large array without formats
defined
r=fromrecords([(2,3.,'abc')]*100000)
it can be slow.
If formats is None, then this will auto-detect formats. Use list of tuples rather than list of lists for
faster processing.

>>> r=np.core.records.fromrecords([(456, 'dbe', 1.2), (2, 'de', 1.3)],
... names='col1,col2,col3')
>>> print r[0]
(456, 'dbe', 1.2)
>>> r.col1
array([456, 2])
>>> r.col2
chararray(['dbe', 'de'],
dtype='|S3')
>>> import cPickle
>>> print cPickle.loads(cPickle.dumps(r))
[(456, 'dbe', 1.2) (2, 'de', 1.3)]

fromstring(datastring, dtype=None, shape=None, offset=0, formats=None, names=None, titles=None, aligned=False, byteorder=None)
create a (read-only) record array from binary data contained in a string

fromfile (fd, dtype=None, shape=None, offset=0, formats=None, names=None, titles=None, aligned=False, byteorder=None)
Create an array from binary file data
If file is a string then that file is opened, else it is assumed to be a file object.

>>> from tempfile import TemporaryFile
>>> a = np.empty(10,dtype='f8,i4,a5')
>>> a[5] = (0.5,10,'abcde')

>>> fd=TemporaryFile()
>>> a = a.newbyteorder('c')
>>> a.tofile(fd)

>>> fd.seek(0)
>>> r=np.core.records.fromfile(fd, formats='f8,i4,a5', shape=10,
... byteorder='<')
>>> print r[5]
(0.5, 10, 'abcde')
>>> r.shape
tuple
```

304 Chapter 3. Routines
3.1.4 Creating character arrays (numpy.char)

Note: numpy.char is the preferred alias for numpy.core.defchararray.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>core.defchararray.array(obj[, itemsize, copy, ...])</code></td>
<td>Create a chararray.</td>
</tr>
<tr>
<td><code>core.defchararray.asarray(obj[, itemsize, unicode, ...])</code></td>
<td>Convert the input to a chararray, copying the data only if necessary.</td>
</tr>
</tbody>
</table>

**array** (obj, itemsize=None, copy=True, unicode=None, order=None)

Create a chararray.

**Parameters**
- **obj**: array of str or unicode-like
- **itemsize**: int, optional
  - `itemsize` is the number of characters per scalar in the resulting array. If `itemsize` is None, and `obj` is an object array or a Python list, the `itemsize` will be automatically determined. If `itemsize` is provided and `obj` is of type str or unicode, then the `obj` string will be chunked into `itemsize` pieces.
- **copy**: bool, optional
  - If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if `obj` is a nested sequence, or if a copy is needed to satisfy any of the other requirements (`itemsize`, `unicode`, `order`, etc.).
- **unicode**: bool, optional
  - When true, the resulting chararray can contain Unicode characters, when false only 8-bit characters. If unicode is None and `obj` is one of the following:
    - a chararray,
    - an ndarray of type str or unicode
    - a Python str or unicode object,
  then the unicode setting of the output array will be automatically determined.
- **order**: {'C', 'F', 'A'}, optional
  - Specify the order of the array. If order is ‘C’ (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is ‘F’, then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is ‘A’, then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

**asarray** (obj, itemsize=None, unicode=None, order=None)

Convert the input to a chararray, copying the data only if necessary.
Versus a regular Numpy array of type *str* or *unicode*, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. +, *, %)

**Parameters**

- **obj** : array of str or unicode-like
- **itemsize** : int, optional

  *itemsize* is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

- **unicode** : bool, optional

  When true, the resulting *chararray* can contain Unicode characters, when false only 8-bit characters. If *unicode* is *None* and *obj* is one of the following:

  - a *chararray*,
  - an ndarray of type *str* or ‘unicode’
  - a Python str or unicode object,

  then the unicode setting of the output array will be automatically determined.

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

  Specify the order of the array. If order is ‘C’ (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is ‘F’, then the returned array will be in Fortran-contiguous order (first-index varies the fastest).

### 3.1.5 Numerical ranges

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arange</code></td>
<td>Return evenly spaced values within a given interval.</td>
</tr>
<tr>
<td><code>linspace</code></td>
<td>Return evenly spaced numbers over a specified interval.</td>
</tr>
<tr>
<td><code>logspace</code></td>
<td>Return numbers spaced evenly on a log scale.</td>
</tr>
<tr>
<td><code>meshgrid</code></td>
<td>Return coordinate matrices from two coordinate vectors.</td>
</tr>
<tr>
<td><code>mgrid</code></td>
<td><em>nd_grid</em> instance which returns a dense multi-dimensional “meshgrid”.</td>
</tr>
<tr>
<td><code>ogrid</code></td>
<td><em>nd_grid</em> instance which returns an open multi-dimensional “meshgrid”.</td>
</tr>
</tbody>
</table>

**arange()**

Return evenly spaced values within a given interval.

Values are generated within the half-open interval [*start*, *stop*) (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in *range* function, but returns a ndarray rather than a list.

**Parameters**

- **start** : number, optional

  Start of interval. The interval includes this value. The default start value is 0.
stop : number
End of interval. The interval does not include this value.

step : number, optional
Spacing between values. For any output out, this is the distance between two adjacent
values, out[i+1] - out[i]. The default step size is 1. If step is specified, start
must also be given.

dtype : dtype
The type of the output array. If dtype is not given, infer the data type from the other
input arguments.

Returns
out : ndarray
Array of evenly spaced values.
For floating point arguments, the length of the result is ceil((stop -
start)/step). Because of floating point overflow, this rule may result in the last
element of out being greater than stop.

See Also:
linspace
Evenly spaced numbers with careful handling of endpoints.
ogrid
Arrays of evenly spaced numbers in N-dimensions
mgrid
Grid-shaped arrays of evenly spaced numbers in N-dimensions

Examples

```python
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0., 1., 2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

linspace (start, stop, num=50, endpoint=True, retstep=False)
Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval [start, stop ].
The endpoint of the interval can optionally be excluded.

Parameters
start : scalar
The starting value of the sequence.

stop : scalar
The end value of the sequence, unless endpoint is set to False. In that case, the sequence
consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded.
Note that the step size changes when endpoint is False.
num : int, optional
    Number of samples to generate. Default is 50.

epsilon endpoint : bool, optional
    If True, stop is the last sample. Otherwise, it is not included. Default is True.

retstep : bool, optional
    If True, return (samples, step), where step is the spacing between samples.

Returns
samples : ndarray
    There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False).

step : float (only if retstep is True)
    Size of spacing between samples.

See Also:

arange
    Similar to linspace, but uses a step size (instead of the number of samples).

logspace
    Samples uniformly distributed in log space.

Examples

>>> np.linspace(2.0, 3.0, num=5)
array([ 2. , 2.25, 2.5 , 2.75, 3. ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([ 2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([ 2. , 2.25, 2.5 , 2.75, 3. ]), 0.25)

Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
>>> plt.plot(x2, y + 0.5, 'o')
>>> plt.ylim([-0.5, 1])
>>> plt.show()
logspace\((start, stop, num=50, endpoint=True, base=10.0)\)

Return numbers spaced evenly on a log scale.

In linear space, the sequence starts at \(base ** start\) (base to the power of \(start\)) and ends with \(base ** stop\) (see \(endpoint\) below).

**Parameters**

*start* : float

\(base ** start\) is the starting value of the sequence.

*stop* : float

\(base ** stop\) is the final value of the sequence, unless \(endpoint\) is False. In that case, \(num + 1\) values are spaced over the interval in log-space, of which all but the last (a sequence of length \(num\)) are returned.

*num* : integer, optional

Number of samples to generate. Default is 50.

*endpoint* : boolean, optional

If true, \(stop\) is the last sample. Otherwise, it is not included. Default is True.

*base* : float, optional

The base of the log space. The step size between the elements in \(\ln(samples) / \ln(base)\) (or \(\log_base(samples)\)) is uniform. Default is 10.0.

**Returns**

*samples* : ndarray

\(num\) samples, equally spaced on a log scale.

**See Also:**

*arange*

Similar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.

*linspace*

Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.
Notes

Logspace is equivalent to the code

```python
>>> y = linspace(start, stop, num=num, endpoint=endpoint)
>>> power(base, y)
```

Examples

```python
>>> np.logspace(2.0, 3.0, num=4)
array([  100. ,   215.443469,   464.15888336,  1000. ])
>>> np.logspace(2.0, 3.0, num=4, endpoint=False)
array([ 100. ,   177.827941 ,  316.22776602,  562.34132519])
>>> np.logspace(2.0, 3.0, num=4, base=2.0)
array([ 4. ,   5.0396842 ,  6.34960421,   8. ])
```

Graphical illustration:

```python
>>> import matplotlib.pyplot as plt

>>> N = 10
>>> x1 = np.logspace(0.1, 1, N, endpoint=True)
>>> x2 = np.logspace(0.1, 1, N, endpoint=False)
>>> y = np.zeros(N)
>>> plt.plot(x1, y, 'o')
>>> plt.plot(x2, y + 0.5, 'o')
>>> plt.ylim([-0.5, 1])
>>> plt.show()
```

```python
meshgrid(x, y)
Return coordinate matrices from two coordinate vectors.

Parameters
x, y : ndarray
  Two 1-D arrays representing the x and y coordinates of a grid.

Returns
X, Y : ndarray
```
For vectors \( x, y \) with lengths \( Nx = \text{len}(x) \) and \( Ny = \text{len}(y) \), return \( X, Y \) where \( X \) and \( Y \) are \((Ny, Nx)\) shaped arrays with the elements of \( x \) and \( y \) repeated to fill the matrix along the first dimension for \( x \), the second for \( y \).

**See Also:**

- `index_tricks.mgrid`
  Construct a multi-dimensional “meshgrid” using indexing notation.

- `index_tricks.ogrid`
  Construct an open multi-dimensional “meshgrid” using indexing notation.

**Examples**

```python
>>> X, Y = np.meshgrid([1, 2, 3], [4, 5, 6, 7])
>>> X
array([[1, 2, 3, 3, 3],
       [1, 2, 3, 3, 3],
       [1, 2, 3, 3, 3],
       [1, 2, 3, 3, 3]])

>>> Y
array([[4, 4, 4, 4, 4],
       [5, 5, 5, 5, 5],
       [6, 6, 6, 6, 6],
       [7, 7, 7, 7, 7]])
```

`meshgrid` is very useful to evaluate functions on a grid.

```python
>>> x = np.arange(-5, 5, 0.1)
>>> y = np.arange(-5, 5, 0.1)
>>> xx, yy = np.meshgrid(x, y)
>>> z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)
```

**mgrid** (sparse=False)

An instance of `numpy.lib.index_tricks.nd_grid` which returns a dense multi-dimensional “meshgrid”.

- nd_grid instance which returns a dense multi-dimensional “meshgrid”.

An instance of `numpy.lib.index_tricks.nd_grid` which returns a dense (or fleshed out) mesh-grid when indexed, so that each returned argument has the same shape. The dimensions and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a `complex number` (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

**Returns**

- mesh-grid ‘ndarrays’ all of the same dimensions:

**See Also:**

- `numpy.lib.index_tricks.nd_grid`
  class of ogrid and mgrid objects

- `ogrid`
  like mgrid but returns open (not fleshed out) mesh grids

- `r_`
  array concatenator
Examples

```python
>>> np.mgrid[0:5,0:5]
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]],
      [[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]]))
```
```
>>> np.mgrid[-1:1:5j]
array([-1. , -0.5, 0. , 0.5, 1. ])
```

**ogrid** *(sparse=False)*

`nd_grid` instance which returns an open multi-dimensional “meshgrid”.

An instance of `numpy.lib.index_tricks.nd_grid` which returns an open (i.e. not fleshed out) meshgrid when indexed, so that only one dimension of each returned array is greater than 1. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a complex number (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

Returns

mesh-grid ‘ndarrays’ with only one dimension :math:`neq 1`:

See Also:

**np.lib.index_tricks.nd_grid**

class of `ogrid` and `mgrid` objects

**mgrid**

like `ogrid` but returns dense (or fleshed out) mesh grids

**r_**

array concatenator

Examples

```python
>>> from numpy import ogrid
```
```
>>> ogrid[-1:1:5j]
array([-1. , -0.5, 0. , 0.5, 1. ])
```
```
>>> ogrid[0:5,0:5]
[array([[0],
       [1],
       [2],
       [3],
       [4]]), array([[0, 1, 2, 3, 4]])]
```
3.1.6 Building matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>diag(v[, k])</code></td>
<td>Extract a diagonal or construct a diagonal array.</td>
</tr>
<tr>
<td><code>diagflat(v[, k])</code></td>
<td>Create a two-dimensional array with the flattened input as a diagonal.</td>
</tr>
<tr>
<td><code>tri(N[, M, k, dtype])</code></td>
<td>Construct an array filled with ones at and below the given diagonal.</td>
</tr>
<tr>
<td><code>tril(m[, k])</code></td>
<td>Lower triangle of an array.</td>
</tr>
<tr>
<td><code>triu(m[, k])</code></td>
<td>Upper triangle of an array.</td>
</tr>
<tr>
<td><code>vander(x[, N])</code></td>
<td>Generate a Van der Monde matrix.</td>
</tr>
</tbody>
</table>

**diag(v, k=0)**

Extract a diagonal or construct a diagonal array.

**Parameters**

- **v**: array_like
  
  If `v` is a 2-D array, return a copy of its `k`-th diagonal. If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th diagonal.

- **k**: int, optional
  
  Diagonal in question. The default is 0. Use `k>0` for diagonals above the main diagonal, and `k` for diagonals below the main diagonal.

**Returns**

- **out**: ndarray
  
  The extracted diagonal or constructed diagonal array.

**See Also:**

- `diagonal`
  
  Return specified diagonals.

- `diagflat`
  
  Create a 2-D array with the flattened input as a diagonal.

- `trace`
  
  Sum along diagonals.

- `triu`
  
  Upper triangle of an array.

- `tril`
  
  Lower triangle of an array.

**Examples**

```python
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.diag(x)
array([0, 4, 8])

>>> np.diag(x, k=1)
array([1, 5])

>>> np.diag(x, k=-1)
array([3, 7])
```
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])

diagflat(v, k=0)
Create a two-dimensional array with the flattened input as a diagonal.

Parameters
v : array_like
    Input data, which is flattened and set as the k-th diagonal of the output.

k : int, optional
    Diagonal to set. The default is 0.

Returns
out : ndarray
    The 2-D output array.

See Also:
diag
    Matlab workalike for 1-D and 2-D arrays.
diagonal
    Return specified diagonals.
trace
    Sum along diagonals.

Examples

>>> np.diagflat([[1,2], [3,4]])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])

>>> np.diagflat([1,2], 1)
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])

tri(N, M=None, k=0, dtype=<type 'float'>)
Construct an array filled with ones at and below the given diagonal.

Parameters
N : int
    Number of rows in the array.

M : int, optional
    Number of columns in the array. By default, M is taken equal to N.

k : int, optional
The sub-diagonal below which the array is filled. \( k = 0 \) is the main diagonal, while \( k < 0 \) is below it, and \( k > 0 \) is above. The default is 0.

**dtype**: dtype, optional
Data type of the returned array. The default is float.

**Returns**
- \( T \) : (N,M) ndarray
  Array with a lower triangle filled with ones, in other words \( T[i, j] == 1 \) for \( i <= j + k \).

**Examples**

```python
>>> np.tri(3, 5, 2, dtype=int)
array([[1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1]])
```

```python
>>> np.tri(3, 5, -1)
array([[ 0., 0., 0., 0., 0.],
       [ 1., 0., 0., 0., 0.],
       [ 1., 1., 0., 0., 0.]])
```

**tril** \((m, k=0)\)
Lower triangle of an array.
Return a copy of an array with elements above the \( k \)-th diagonal zeroed.

**Parameters**
- \( m \) : array_like, shape (M, N)
  Input array.
- \( k \) : int
  Diagonal above which to zero elements. \( k = 0 \) is the main diagonal, \( k \) is below it and \( k > 0 \) is above.

**Returns**
- \( L \) : ndarray, shape (M, N)
  Lower triangle of \( m \), of same shape and data-type as \( m \).

**See Also:**
- triu

**Examples**

```python
>>> np.tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 0., 0., 0.],
       [ 4., 0., 0.],
       [ 7., 8., 0.],
       [10., 11., 12.]])
```

**triu** \((m, k=0)\)
Upper triangle of an array.
Construct a copy of a matrix with elements below the \( k \)-th diagonal zeroed.

3.1. Array creation routines
Please refer to the documentation for `tril`.

**See Also:**

`tril`

**Examples**

```python
>>> np.triu([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]], -1)
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 0,  8,  9],
       [ 0,  0, 12]])
```

---

**vander** *(x, N=None)*

Generate a Van der Monde matrix.

The columns of the output matrix are decreasing powers of the input vector. Specifically, the i-th output column is the input vector to the power of \( N - i - 1 \). Such a matrix with a geometric progression in each row is named Van Der Monde, or Vandermonde matrix, from Alexandre-Theophile Vandermonde.

**Parameters**

- **x**: array_like
  - 1-D input array.

- **N**: int, optional
  - Order of (number of columns in) the output. If \( N \) is not specified, a square array is returned (\( N = \text{len}(x) \)).

**Returns**

- **out**: ndarray
  - Van der Monde matrix of order \( N \). The first column is \( x^{(N-1)} \), the second \( x^{(N-2)} \) and so forth.

**References**

[R144]

**Examples**

```python
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1,  1],
       [ 4,  2,  1,  1],
       [ 9,  3,  1,  1],
       [25,  5,  1,  1]])
```

```python
>>> np.column_stack([x**(N-i) for i in range(N)])
array([[ 1,  1,  1,  1],
       [ 4,  2,  1,  1],
       [ 9,  3,  1,  1],
       [25,  5,  1,  1]])
```

```python
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[ 1,  1,  1,  1,  1],
       [ 4,  2,  1,  1,  1],
       [ 9,  3,  1,  1,  1],
       [25,  5,  1,  1,  1]])
```
[ 8, 4, 2, 1],
[ 27, 9, 3, 1],
[125, 25, 5, 1])

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```python
>>> np.linalg.det(np.vander(x))
48.000000000000043
>>> (5-3) * (5-2) * (5-1) * (3-2) * (3-1) * (2-1)
48
```

3.1.7 The Matrix class

<table>
<thead>
<tr>
<th><code>mat(data[, dtype])</code></th>
<th>Interpret the input as a matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bmat(obj[, ldict, gdict])</code></td>
<td>Build a matrix object from a string, nested sequence, or array.</td>
</tr>
</tbody>
</table>

**`mat(data, dtype=None)`**

Interpret the input as a matrix.

Unlike `matrix`, `asmatrix` does not make a copy if the input is already a matrix or an ndarray. Equivalent to `matrix(data, copy=False)`.

**Parameters**

- `data`: array_like
  - Input data.

**Returns**

- `mat`: matrix
  - `data` interpreted as a matrix.

**Examples**

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

>>> m = np.asmatrix(x)

>>> x[0, 0] = 5

>>> m
matrix([[5, 2],
         [3, 4]])
```

**`bmat(obj[, ldict, gdict])`**

Build a matrix object from a string, nested sequence, or array.

**Parameters**

- `obj`: string, sequence or array
  - Input data. Variables names in the current scope may be referenced, even if `obj` is a string.

**Returns**

- `out`: matrix
Returns a matrix object, which is a specialized 2-D array.

See Also:

matrix

Examples

```python
>>> A = np.mat('1 1; 1 1')
>>> B = np.mat('2 2; 2 2')
>>> C = np.mat('3 4; 5 6')
>>> D = np.mat('7 8; 9 0')

All the following expressions construct the same block matrix:

```python

```python
np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
np.bmat('A,B; C,D')
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```
Returns

```
reshaped_array : ndarray
```

This will be a new view object if possible; otherwise, it will be a copy.

See Also:

```
ndarray.reshape
```

Equivalent method.

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
>>> np.reshape(a, (3,-1))  # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
ravel (a, order='C')
```

Return a flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Parameters

```
a : array_like
```

Input array. The elements in `a` are read in the order specified by `order`, and packed as a
1-D array.

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

The elements of `a` are read in this order. It can be either ‘C’ for row-major order, or `F`
for column-major order. By default, row-major order is used.

Returns

```
1d_array : ndarray
```

Output of the same dtype as `a`, and of shape `(a.size(),)`.

See Also:

```
ndarray.flat
```

1-D iterator over an array.

```
ndarray.flatten
```

1-D array copy of the elements of an array in row-major order.

Notes

In row-major order, the row index varies the slowest, and the column index the quickest. This can be generalized
to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the
index along the last quickest. The opposite holds for Fortran-, or column-major, mode.
Examples

If an array is in C-order (default), then ravel is equivalent to reshape(-1):

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print x.reshape(-1)
[1 2 3 4 5 6]

>>> print np.ravel(x)
[1 2 3 4 5 6]
```

When flattening using Fortran-order, however, we see

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

**flat**

A 1-D iterator over the array.

This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

**flatten**

Return a copy of the array collapsed into one dimension.

**flatiter**

Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])

>>> x.flat[3]
4

>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])

>>> x.T.flat[3]
5

>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])

>>> x.flat[[1, 4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

**flatten()**

Return a copy of the array collapsed into one dimension.
Parameters

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

Whether to flatten in C (row-major) or Fortran (column-major) order. The default is 'C'.

Returns

y : ndarray

A copy of the input array, flattened to one dimension.

See Also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

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

3.2.2 Transpose-like operations

<table>
<thead>
<tr>
<th>rollaxis (a, axis[, start])</th>
<th>Roll the specified axis backwards, until it lies in a given position.</th>
</tr>
</thead>
<tbody>
<tr>
<td>swapaxes (a, axis1, axis2)</td>
<td>Interchange two axes of an array.</td>
</tr>
<tr>
<td>ndarray.T</td>
<td>Same as self.transpose(), except that self is returned if self.ndim &lt; 2.</td>
</tr>
<tr>
<td>transpose (a[, axes])</td>
<td>Permute the dimensions of an array.</td>
</tr>
</tbody>
</table>

rollaxis (a, axis, start=0)
Roll the specified axis backwards, until it lies in a given position.

Parameters

a : ndarray
Input array.

axis : int
The axis to roll backwards. The positions of the other axes do not change relative to one another.

start : int, optional
The axis is rolled until it lies before this position.

Returns

res : ndarray
Output array.

See Also:

roll
Roll the elements of an array by a number of positions along a given axis.

3.2. Array manipulation routines
Examples

```python
>>> a = np.ones((3,4,5,6))
>>> np.rollaxis(a, 3, 1).shape
(3, 6, 4, 5)
>>> np.rollaxis(a, 2).shape
(5, 3, 4, 6)
>>> np.rollaxis(a, 1, 4).shape
(3, 5, 6, 4)
```

**swapaxes** (*a*, *axis1*, *axis2*)
Interchange two axes of an array.

**Parameters**
- *a*: array_like
  - Input array.
- *axis1*: int
  - First axis.
- *axis2*: int
  - Second axis.

**Returns**
- *a_swapped*: ndarray
  - If *a* is an ndarray, then a view of *a* is returned; otherwise a new array is created.

Examples

```python
>>> x = np.array([[1, 2, 3]])
>>> np.swapaxes(x, 0, 1)
array([[1],
       [2],
       [3]])

>>> x = np.array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])
>>> x
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])

>>> np.swapaxes(x, 0, 2)
array([[[0, 4],
        [2, 6]],
       [[1, 5],
        [3, 7]]])
```

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

Examples

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

```nim
np.array([1., 2., 3., 4.])  
np.array([ 1.,  2.,  3.,  4.])  
np.array([ 1.,  2.,  3.,  4.])
```

```
transpose (a, axes=None)
```
Permute the dimensions of an array.

**Parameters**
- `a`: array_like
  Input array.
- `axes`: list of ints, optional
  By default, reverse the dimensions, otherwise permute the axes according to the values given.

**Returns**
- `p`: ndarray
  `a` with its axes permuted. A view is returned whenever possible.

**See Also:**
- `rollaxis`

**Examples**

```nim
np.arange(4).reshape((2,2))
```

```nim
np.transpose(x)
```

```nim
np.transpose(x, (1, 0, 2)).shape
```

```
3.2.3 Changing number of dimensions
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atleast_1d(*arys)</code></td>
<td>Convert inputs to arrays with at least one dimension.</td>
</tr>
<tr>
<td><code>atleast_2d(*arys)</code></td>
<td>View inputs as arrays with at least two dimensions.</td>
</tr>
<tr>
<td><code>atleast_3d(*arys)</code></td>
<td>View inputs as arrays with at least three dimensions.</td>
</tr>
<tr>
<td><code>broadcast</code></td>
<td>Produce an object that mimics broadcasting.</td>
</tr>
<tr>
<td><code>broadcast_arrays(*arys)</code></td>
<td>Broadcast any number of arrays against each other.</td>
</tr>
<tr>
<td><code>expand_dims(a, axis)</code></td>
<td>Expand the shape of an array.</td>
</tr>
<tr>
<td><code>squeeze(a)</code></td>
<td>Remove single-dimensional entries from the shape of an array.</td>
</tr>
</tbody>
</table>

**3.2. Array manipulation routines**
**atleast_1d**(*arys*)  
Convert inputs to arrays with at least one dimension.  
Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**  
array1, array2, ... : array_like  
One or more input arrays.

**Returns**  
ret : ndarray  
An array, or sequence of arrays, each with a.ndim >= 1. Copies are made only if necessary.

**See Also:**  
*atleast_2d*, *atleast_3d*

**Examples**

```python  
>>> np.atleast_1d(1.0)  
array([ 1.])  

>>> x = np.arange(9.0).reshape(3,3)  
>>> np.atleast_1d(x)  
array([[ 0., 1., 2.],  
       [ 3., 4., 5.],  
       [ 6., 7., 8.]])  
>>> np.atleast_1d(x) is x  
True  

>>> np.atleast_1d(1, [3, 4])  
[array([1]), array([3, 4])]  
```

**atleast_2d**(*arys*)  
View inputs as arrays with at least two dimensions.

**Parameters**  
array1, array2, ... : array_like  
One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns**  
res, res2, ... : ndarray  
An array, or tuple of arrays, each with a.ndim >= 2. Copies are avoided where possible, and views with two or more dimensions are returned.

**See Also:**  
*atleast_1d*, *atleast_3d*

**Examples**

```python  
>>> np.atleast_2d(3.0)  
array([[ 3.]])
>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[ 0.,  1.,  2.]])
>>> np.atleast_2d(x).base is x
True

>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]

atleast_3d(*arys)
View inputs as arrays with at least three dimensions.

Parameters
array1, array2, ... : array_like
One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

Returns
res1, res2, ... : ndarray
An array, or tuple of arrays, each with a.ndim >= 3. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape N becomes a view of shape (1, N, 1). A 2-D array of shape (M, N) becomes a view of shape (M, N, 1).

See Also:
atleast_1d, atleast_2d

Examples

>>> np.atleast_3d(3.0)
array([[ 3.]])
>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)

>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x
True

>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...    print arr, arr.shape
...
[[1]
 [2]] (1, 2, 1)
[[[1]
 [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)

class broadcast()
Produce an object that mimics broadcasting.

3.2. Array manipulation routines
Parameters

\texttt{in1, in2, ...} : array_like

Input parameters.

Returns

\texttt{b} : broadcast object

Broadcast the input parameters against one another, and return an object that encapsulates the result. Amongst others, it has \texttt{shape} and \texttt{nd} properties, and may be used as an iterator.

Examples

Manually adding two vectors, using broadcasting:

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

>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]

array([[ 5.,  6.,  7.],
        [ 6.,  7.,  8.],
        [ 7.,  8.,  9.]])
\end{verbatim}

Compare against built-in broadcasting:

\begin{verbatim}
>>> x + y
array([[5, 6, 7],
        [6, 7, 8],
        [7, 8, 9]])
\end{verbatim}

Methods

\texttt{broadcast_arrays(*args)}

Broadcast any number of arrays against each other.

Parameters

\texttt{‘*args’ : arrays}

The arrays to broadcast.

Returns

\texttt{broadcasted} : list of arrays

These arrays are views on the original arrays. They are typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location. If you need to write to the arrays, make copies first.

Examples

\begin{verbatim}
>>> x = np.array([[1,2,3]])
>>> y = np.array([[1],[2],[3]])
>>> np.broadcast_arrays(x, y)
[array([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]],
        dtype=int64), array([[1],
        [2],
        [3]],
        dtype=int64)]
\end{verbatim}
Here is a useful idiom for getting contiguous copies instead of non-contiguous views.

```python
>>> map(np.array, np.broadcast_arrays(x, y))
[array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]),
  array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])]
```

**expand_dims** *(a, axis)*

Expand the shape of an array.

Insert a new axis, corresponding to a given position in the array shape.

**Parameters**

- **a**: array_like
  - Input array.
- **axis**: int
  - Position (amongst axes) where new axis is to be inserted.

**Returns**

- **res**: ndarray
  - Output array. The number of dimensions is one greater than that of the input array.

**See Also:**

doc.indexing, atleast_1d, atleast_2d, atleast_3d

**Examples**

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

The following is equivalent to `x[np.newaxis, :]` or `x[np.newaxis]`:

```python
>>> y = np.expand_dims(x, axis=0)
```  
```python
>>> y
array([[1, 2]])
```  
```python
>>> y.shape
(1, 2)
```

```python
>>> y = np.expand_dims(x, axis=1)  # Equivalent to `x[:, np.newaxis]`
>>> y
array([[1],
       [2]])
>>> y.shape
(2, 1)
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects.
>>> np.newaxis is None
True

**squeeze** (*a*)
Remove single-dimensional entries from the shape of an array.

**Parameters**
- **a**: array_like
  Input data.

**Returns**
- **squeezed**: ndarray
  The input array, but with all dimensions of length 1 removed. Whenever possible, a view on *a* is returned.

**Examples**

```python
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
```

### 3.2.4 Changing kind of array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asarray</td>
<td>Convert the input to an array.</td>
</tr>
<tr>
<td>asanyarray</td>
<td>Convert the input to a ndarray, but pass ndarray subclasses through.</td>
</tr>
<tr>
<td>asmatrix</td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td>asfarray</td>
<td>Return an array converted to a float type.</td>
</tr>
<tr>
<td>asfortranarray</td>
<td>Return an array laid out in Fortran order in memory.</td>
</tr>
<tr>
<td>asscalar</td>
<td>Convert an array of size 1 to its scalar equivalent.</td>
</tr>
<tr>
<td>require</td>
<td>Return an ndarray of the provided type that satisfies requirements.</td>
</tr>
</tbody>
</table>

**asarray** (*a*, *dtype=None*, *order=None*)
Convert the input to an array.

**Parameters**
- **a**: array_like
  Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.

- **dtype**: data-type, optional
  By default, the data-type is inferred from the input data.

- **order**: {'C', 'F'}, optional
  Whether to use row-major ('C') or column-major ('F' for FORTRAN) memory representation. Defaults to ‘C’.

**Returns**
- **out**: ndarray
  Array interpretation of *a*. No copy is performed if the input is already an ndarray. If *a* is a subclass of ndarray, a base class ndarray is returned.
See Also:

**asanyarray**
Similar function which passes through subclasses.

**ascontiguousarray**
Convert input to a contiguous array.

**asfarray**
Convert input to a floating point ndarray.

**asfortranarray**
Convert input to an ndarray with column-major memory order.

**asarray_chkfinite**
Similar function which checks input for NaNs and Infs.

**fromiter**
Create an array from an iterator.

**fromfunction**
Construct an array by executing a function on grid positions.

Examples
Convert a list into an array:

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

Existing arrays are not copied:

```python
>>> a = np.array([1, 2])
>>> np.asarray(a)
is a True
```

Contrary to `asanyarray`, ndarray subclasses are not passed through:

```python
>>> issubclass(np.matrix, np.ndarray)
True
>>> a = np.matrix([[1, 2]])
>>> np.asarray(a) is a False
>>> np.asanyarray(a) is a True
```

**asanyarray** (`a, dtype=None, order=None`)
Convert the input to a ndarray, but pass ndarray subclasses through.

Parameters
- **a** : array_like
  Input data, in any form that can be converted to an array. This includes scalars, lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.
- **dtype** : data-type, optional
  By default, the data-type is inferred from the input data.
- **order** : {'C', 'F'}, optional
Whether to use row-major (‘C’) or column-major (‘F’) memory representation. Defaults to ‘C’.

**Returns**

- **out**: ndarray or an ndarray subclass
  
  Array interpretation of *a*. If *a* is an ndarray or a subclass of ndarray, it is returned as-is and no copy is performed.

**See Also:**

- **asarray**: Similar function which always returns ndarrays.
- **ascontiguousarray**: Convert input to a contiguous array.
- **asfarray**: Convert input to a floating point ndarray.
- **asfortranarray**: Convert input to an ndarray with column-major memory order.
- **asarray_chkfinite**: Similar function which checks input for NaNs and Infs.
- **fromiter**: Create an array from an iterator.
- **fromfunction**: Construct an array by executing a function on grid positions.

**Examples**

Convert a list into an array:

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

Instances of *ndarray* subclasses are passed through as-is:

```python
>>> a = np.matrix([[1, 2]])
>>> np.asanyarray(a) is a
True
```

**asmatrix**(data, dtype=None)

Interpret the input as a matrix.

Unlike *matrix*, *asmatrix* does not make a copy if the input is already a matrix or an ndarray. Equivalent to *matrix*(data, copy=False).

**Parameters**

- **data**: array_like
  
  Input data.

**Returns**

- **mat**: matrix
  
  *data* interpreted as a matrix.
Examples

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

>>> m = np.asmatrix(x)

>>> x[0,0] = 5

>>> m
matrix([[5, 2],
        [3, 4]])
```

**asfarray** *(a, dtype='numpy.float64')*

Return an array converted to a float type.

**Parameters**

- `a`: array_like
  - The input array.
- `dtype`: str or dtype object, optional
  - Float type code to coerce input array `a`. If `dtype` is one of the ‘int’ dtypes, it is replaced with float64.

**Returns**

- `out`: ndarray
  - The input `a` as a float ndarray.

Examples

```python
>>> np.asfarray([2, 3])
array([ 2., 3.])

>>> np.asfarray([2, 3], dtype='float')
array([ 2., 3.])

>>> np.asfarray([2, 3], dtype='int8')
array([ 2., 3.])
```

**asfortranarray** *(a, dtype=None)*

Return an array laid out in Fortran order in memory.

**Parameters**

- `a`: array_like
  - Input array.
- `dtype`: str or dtype object, optional
  - By default, the data-type is inferred from the input data.

**Returns**

- `out`: ndarray
  - The input `a` in Fortran, or column-major, order.

**See Also:**

- `ascontiguousarray`
  - Convert input to a contiguous (C order) array.
**asanyarray**
Convert input to an ndarray with either row or column-major memory order.

**require**
Return an ndarray that satisfies requirements.

**ndarray.flags**
Information about the memory layout of the array.

**Examples**

```python
given
```n
```python
>>> x = np.arange(6).reshape(2,3)
>>> y = np.asfortranarray(x)
>>> x.flags['F_CONTIGUOUS']
False
>>> y.flags['F_CONTIGUOUS']
True
```

**asscalar (a)**
Convert an array of size 1 to its scalar equivalent.

**Parameters**
- **a**: ndarray
  Input array of size 1.

**Returns**
- **out**: scalar
  Scalar representation of `a`. The input data type is preserved.

**Examples**

```python
>>> np.asscalar(np.array([24]))
24
```

**require (a, dtype=None, requirements=None)**
Return an ndarray of the provided type that satisfies requirements.

This function is useful to be sure that an array with the correct flags is returned for passing to compiled code (perhaps through ctypes).

**Parameters**
- **a**: array_like
  The object to be converted to a type-and-requirement-satisfying array.
- **dtype**: data-type
  The required data-type, the default data-type is float64).
- **requirements**: str or list of str
  The requirements list can be any of the following
  - ‘F_CONTIGUOUS’ (‘F’) - ensure a Fortran-contiguous array
  - ‘C_CONTIGUOUS’ (‘C’) - ensure a C-contiguous array
  - ‘ALIGNED’ (‘A’) - ensure a data-type aligned array
  - ‘WRITEABLE’ (‘W’) - ensure a writable array
  - ‘OWNDATA’ (‘O’) - ensure an array that owns its own data
See Also:

asarray
Convert input to an ndarray.

asanyarray
Convert to an ndarray, but pass through ndarray subclasses.

ascontiguousarray
Convert input to a contiguous array.

asfortranarray
Convert input to an ndarray with column-major memory order.

ndarray.flags
Information about the memory layout of the array.

Notes
The returned array will be guaranteed to have the listed requirements by making a copy if needed.

Examples

```python
>>> x = np.arange(6).reshape(2,3)
>>> x.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

>>> y = np.require(x, dtype=np.float32, requirements=['A', 'O', 'W', 'F'])
>>> y.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

3.2.5 Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column_stack</td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td>concatenate</td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td>dstack</td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td>hstack</td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td>vstack</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

**column_stack** *(tup)*

Stack 1-D arrays as columns into a 2-D array.

**Parameters**

- **tup**: sequence of 1-D or 2-D arrays.

3.2. Array manipulation routines 333
Arrays to stack. All of them must have the same first dimension.

Returns
stacked : 2-D array

The array formed by stacking the given arrays.

See Also:

hstack, vstack, concatenate

Notes

This function is equivalent to np.vstack(tup).T.

Examples

>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

concatenate ()

Join a sequence of arrays together.

Parameters

a1, a2, ... : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to axis
(the first, by default).

axis : int, optional

The axis along which the arrays will be joined. Default is 0.

Returns

res : ndarray

The concatenated array.

See Also:

ma.concatenate

Concatenate function that preserves input masks.

array_split

Split an array into multiple sub-arrays of equal or near-equal size.

split

Split array into a list of multiple sub-arrays of equal size.

hsplit

Split array into multiple sub-arrays horizontally (column wise)

vsplit

Split array into multiple sub-arrays vertically (row wise)

dsplit

Split array into multiple sub-arrays along the 3rd axis (depth).

hstack

Stack arrays in sequence horizontally (column wise)
vstack
Stack arrays in sequence vertically (row wise)

dstack
Stack arrays in sequence depth wise (along third dimension)

Notes
When one or more of the arrays to be concatenated is a MaskedArray, this function will return a MaskedArray object instead of an ndarray, but the input masks are not preserved. In cases where a MaskedArray is expected as input, use the ma.concatenate function from the masked array module instead.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
```

This function will not preserve masking of MaskedArray inputs.

```python
>>> a = np.ma.arange(3)
>>> a[1] = np.ma.masked
>>> b = np.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
            fill_value = 999999)
>>> b
array([2, 3, 4])
>>> np.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
            fill_value = 999999)
>>> np.ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
            fill_value = 999999)
```

dstack (tup)
Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by dsplit. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

Parameters
tup : sequence of arrays

Arrays to stack. All of them must have the same shape along all but the third axis.

Returns
stacked : ndarray

The array formed by stacking the given arrays.
See Also:

vstack
Stack along first axis.

hstack
Stack along second axis.

concatenate
Join arrays.

dsplit
Split array along third axis.

Notes

Equivalent to np.concatenate(tup, axis=2)

Examples

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

>>> a = np.array([[1],[2],[3]])
g >>> b = np.array([[2],[3],[4]])
g >>> np.dstack((a,b))
g array([[1, 2],
        [2, 3],
        [3, 4]])
```

hstack(tup)
Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

Parameters

tup : sequence of ndarrays

All arrays must have the same shape along all but the second axis.

Returns

stacked : ndarray

The array formed by stacking the given arrays.

See Also:

vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third axis).

concatenate
Join a sequence of arrays together.
**hsplit**

Split array along second axis.

**Notes**

Equivalent to `np.concatenate(tup, axis=1)`

**Examples**

```python
equiv
equiv
equiv
equiv
```

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

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

**vstack** *(tup)*

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by `vsplit`.

**Parameters**

- **tup**: sequence of ndarrays

  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

**Returns**

- **stacked**: ndarray

  The array formed by stacking the given arrays.

**See Also:**

- **hstack**
  Stack arrays in sequence horizontally (column wise).

- **dstack**
  Stack arrays in sequence depth wise (along third dimension).

- **concatenate**
  Join a sequence of arrays together.

- **vsplit**
  Split array into a list of multiple sub-arrays vertically.

**Notes**

Equivalent to `np.concatenate(tup, axis=0)`

**Examples**

```python
equiv
equiv
equiv
equiv
```

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

### 3.2.6 Splitting arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array_split</code></td>
<td>Split an array into multiple sub-arrays of equal or near-equal size.</td>
</tr>
<tr>
<td><code>split</code></td>
<td>Split an array into multiple sub-arrays of equal size.</td>
</tr>
<tr>
<td><code>dsplit</code></td>
<td>Split array into multiple sub-arrays along the 3rd axis (depth).</td>
</tr>
<tr>
<td><code>hsplit</code></td>
<td>Split an array into multiple sub-arrays horizontally (column-wise).</td>
</tr>
<tr>
<td><code>split</code></td>
<td>Split an array into multiple sub-arrays of equal size.</td>
</tr>
<tr>
<td><code>vsplit</code></td>
<td>Split an array into multiple sub-arrays vertically (row-wise).</td>
</tr>
</tbody>
</table>

**`array_split`** *(ary, indices_or_sections, axis=0)*

Split an array into multiple sub-arrays of equal or near-equal size.

Please refer to the `split` documentation. The only difference between these functions is that `array_split` allows `indices_or_sections` to be an integer that does not equally divide the axis.

**See Also:**

`split`

Split array into multiple sub-arrays of equal size.

**Examples**

```python
>>> x = np.arange(8.0)
>>> np.array_split(x, 3)
[array([ 0.,  1.,  2.]), array([ 3.,  4.,  5.]), array([ 6.,  7.])]
```

**`dsplit`** *(ary, indices_or_sections)*

Split array into multiple sub-arrays along the 3rd axis (depth).

Please refer to the `split` documentation. `dsplit` is equivalent to `split` with `axis=2`, the array is always split along the third axis provided the array dimension is greater than or equal to 3.

**See Also:**

`split`

Split an array into multiple sub-arrays of equal size.

**Examples**

```python
>>> x = np.arange(16.0).reshape(2, 2, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```
>>> np.dsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
 array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]]))

>>> np.dsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
 array([[ 3.],
        [ 7.],
        [11.],
        [15.]]),
 array([], dtype=float64)]

**hsplit** *(ary, indices_or_sections)*
Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the `split` documentation. `hsplit` is equivalent to `split` with `axis=1`, the array is always split along the second axis regardless of the array dimension.

**See Also:**

**split**
Split an array into multiple sub-arrays of equal size.

**Examples**

```python
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])

>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
 array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]]))

>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
 array([[ 3.],
        [ 7.],
        [11.],
        [15.]]),
 array([], dtype=float64)]
```
With a higher dimensional array the split is still along the second axis.

```python
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
>>>
>>> np.hsplit(x, 2)
[array([[[ 0.,  1.],
         [ 4.,  5.]],
        [[ 2.,  3.],
         [ 6.,  7.]]])
```

**split** *(ary, indices_or_sections, axis=0)*

Split an array into multiple sub-arrays of equal size.

**Parameters**
- **ary**: ndarray
  - Array to be divided into sub-arrays.
- **indices_or_sections**: int or 1-D array
  - If `indices_or_sections` is an integer, `N`, the array will be divided into `N` equal arrays along `axis`. If such a split is not possible, an error is raised.
  - If `indices_or_sections` is a 1-D array of sorted integers, the entries indicate where along `axis` the array is split. For example, `[2, 3]` would, for `axis = 0`, result in
    - `ary[2:]`
    - `ary[2:3]`
    - `ary[3:]`
  - If an index exceeds the dimension of the array along `axis`, an empty sub-array is returned correspondingly.
- **axis**: int, optional
  - The axis along which to split, default is 0.

**Returns**
- **sub-arrays**: list of ndarrays
  - A list of sub-arrays.

**Raises**
- **ValueError**
  - If `indices_or_sections` is given as an integer, but a split does not result in equal division.

**See Also**:
- **array_split**
  - Split an array into multiple sub-arrays of equal or near-equal size. Does not raise an exception if an equal division cannot be made.
- **hsplit**
  - Split array into multiple sub-arrays horizontally (column-wise).
- **vsplit**
  - Split array into multiple sub-arrays vertically (row wise).
dsplit
Split array into multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

Examples

```python
>>> x = np.arange(9.0)
>>> np.split(x, 3)
[array([ 0., 1., 2.]), array([ 3., 4., 5.]), array([ 6., 7., 8.])]

>>> x = np.arange(8.0)
>>> np.split(x, [3, 5, 6, 10])
[array([ 0., 1., 2.]),
 array([ 3., 4.]),
 array([ 5.]),
 array([ 6., 7.]),
 array([], dtype=float64)]
```

vsplit
Split an array into multiple sub-arrays vertically (row-wise).

Please refer to the `split` documentation. **vspl** is equivalent to `split` with `axis=0` (default), the array is always split along the first axis regardless of the array dimension.

See Also:

**split**
Split an array into multiple sub-arrays of equal size.

Examples

```python
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]]

>>> np.vsplit(x, 2)
[array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.]]),
 array([[ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]]])

>>> np.vsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.]],
       array([[ 8.,  9., 10., 11.],
               [12., 13., 14., 15.]]))
```
array([[ 12.,  13.,  14.,  15.]]),
array([], dtype=float64))

With a higher dimensional array the split is still along the first axis.

```python
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[ 0.,  1.],
         [ 2.,  3.]],
       [[ 4.,  5.],
         [ 6.,  7.]]])
>>> np.vsplit(x, 2)
[array([[[ 0.,  1.],
         [ 2.,  3.]],
       [[ 4.,  5.],
         [ 6.,  7.]]])
```

### 3.2.7 Tiling arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tile(A, reps)</code></td>
<td>Construct an array by repeating A the number of times given by reps.</td>
</tr>
<tr>
<td><code>repeat(a, repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
</tbody>
</table>

**tile** *(A, reps)*

Construct an array by repeating A the number of times given by reps.

- If `reps` has length `d`, the result will have dimension of \( \max(d, A\.ndim) \).
- If `A\.ndim < d`, A is promoted to be `d`-dimensional by prepending new axes. So a shape `(3,)` array is promoted to `(1, 3)` for 2-D replication, or shape `(1, 1, 3)` for 3-D replication. If this is not the desired behavior, promote A to `d`-dimensions manually before calling this function.
- If `A\.ndim > d`, `reps` is promoted to `A\.ndim` by pre-pending 1’s to it. Thus for an `A` of shape `(2, 3, 4, 5)`, a `reps` of `(2, 2)` is treated as `(1, 1, 2, 2)`.

**Parameters**

- `A`: array_like
  - The input array.
- `reps`: array_like
  - The number of repetitions of `A` along each axis.

**Returns**

- `c`: ndarray
  - The tiled output array.

**See Also:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>repeat</code></td>
<td>Repeat elements of an array.</td>
</tr>
</tbody>
</table>

**Examples**

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

>>> b = np.array([[1, 2], [3, 4]])
>>> np.tile(b, 2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> np.tile(b, (2, 1))
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])

repeat (a, repeats, axis=None)
Repeat elements of an array.

Parameters
a : array_like
   Input array.
repeats : {int, array of ints}
   The number of repetitions for each element. repeats is broadcasted to fit
   the shape of the given axis.
axis : int, optional
   The axis along which to repeat values. By default, use the flattened input
   array, and return a flat output array.

Returns
repeated_array : ndarray
   Output array which has the same shape as a, except along the given axis.

See Also:
tile
   Tile an array.

Examples

>>> x = np.array([[1,2],[3,4]])
>>> np.repeat(x, 2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 2, 2, 3, 3, 4, 4, 4]],
       [3, 3, 3, 4, 4, 4]])
>>> np.repeat(x, [1, 2], axis=0)
array([[1, 2],
       [3, 4],
       [3, 4]])
3.2.8 Adding and removing elements

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>delete</code></td>
<td>Return a new array with sub-arrays along an axis deleted.</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>Insert values along the given axis before the given indices.</td>
</tr>
<tr>
<td><code>append</code></td>
<td>Append values to the end of an array.</td>
</tr>
<tr>
<td><code>resize</code></td>
<td>Return a new array with the specified shape.</td>
</tr>
<tr>
<td><code>trim_zeros</code></td>
<td>Trim the leading and/or trailing zeros from a 1-D array or sequence.</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>Find the unique elements of an array.</td>
</tr>
</tbody>
</table>

**delete**(arr, obj[, axis])

Return a new array with sub-arrays along an axis deleted.

**Parameters**

* arr : array_like
  Input array.

* obj : slice, int or array of ints
  Indicate which sub-arrays to remove.

* axis : int, optional
  The axis along which to delete the subarray defined by *obj*. If *axis* is None, *obj* is applied to the flattened array.

**Returns**

* out : ndarray
  A copy of *arr* with the elements specified by *obj* removed. Note that *delete* does not occur in-place. If *axis* is None, *out* is a flattened array.

**See Also:**

* insert
  Insert elements into an array.

* append
  Append elements at the end of an array.

**Examples**

```python
>>> arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> arr
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12]])
>>> np.delete(arr, 1, 0)
array([[1, 2, 3, 4],
       [9, 10, 11, 12]])

>>> np.delete(arr, np.s_[::2], 1)
array([[2, 4],
       [6, 8],
       [10, 12]])

>>> np.delete(arr, [1,3,5], None)
array([[1, 3, 5, 7, 8, 9, 10, 11, 12]])
```

**insert**(arr, obj, values[, axis])

Insert values along the given axis before the given indices.
Parameters

arr : array_like
Input array.

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

values : array_like
Values to insert into arr. If the type of values is different from that of arr, values is converted to the type of arr.

axis : int, optional
Axis along which to insert values. If axis is None then arr is flattened first.

Returns

out : ndarray
A copy of arr with values inserted. Note that insert does not occur in-place: a new array is returned. If axis is None, out is a flattened array.

See Also:

append
Append elements at the end of an array.

delete
Delete elements from an array.

Examples

```python
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])

>>> b = a.flatten()
>>> b
array([1, 1, 2, 2, 3, 3])
>>> np.insert(b, [2, 2], [5, 6])
array([1, 1, 5, 6, 2, 2, 3, 3])
>>> np.insert(b, slice(2, 4), [5, 6])
array([1, 1, 5, 2, 6, 2, 3, 3])
>>> np.insert(b, [2, 2], [7.13, False]) # type casting
array([1, 1, 7, 0, 2, 2, 3, 3])
```
>>> x = np.arange(8).reshape(2, 4)
>>> idx = (1, 3)
>>> np.insert(x, idx, 999, axis=1)
array([[ 0, 999,  1,  2, 999,  3],
       [ 4, 999,  5,  6, 999,  7]])

append(arr, values, axis=None)
Append values to the end of an array.

Parameters
   arr : array_like
       Values are appended to a copy of this array.
   values : array_like
       These values are appended to a copy of arr. It must be of the correct shape (the same shape as arr, excluding axis). If axis is not specified, values can be any shape and will be flattened before use.
   axis : int, optional
       The axis along which values are appended. If axis is not given, both arr and values are flattened before use.

Returns
   out : ndarray
       A copy of arr with values appended to axis. Note that append does not occur in-place: a new array is allocated and filled. If axis is None, out is a flattened array.

See Also:
   insert
       Insert elements into an array.
   delete
       Delete elements from an array.

Examples

>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

When axis is specified, values must have the correct shape.

>>> np.append([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[[1, 2, 3], [4, 5, 6]],
       [[7, 8, 9]]])
>>> np.append([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
Traceback (most recent call last):
  ...
ValueError: arrays must have same number of dimension

resize(a, new_shape)
Return a new array with the specified shape.

If the new array is larger than the original array, then the new array is filled with repeated copied of a. Note that this behavior is different from a.resize(new_shape) which fills with zeros instead of repeated copies of a.
Parameters
   a : array_like
       Array to be resized.
   new_shape : {tuple, int}
       Shape of resized array.

Returns
   reshaped_array : ndarray
       The new array is formed from the data in the old array, repeated if necessary to fill out
       the required number of elements. The data are repeated in the order that the data are
       stored in memory.

See Also:
   ndarray.resize
       resize an array in-place.

Examples

>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(1,4))
aarray([[0, 1, 2, 3]])
>>> np.resize(a,(2,4))
aarray([[0, 1, 2, 3],
       [0, 1, 2, 3]])

trim_zeros(filt, trim='fb')
   Trim the leading and/or trailing zeros from a 1-D array or sequence.

Parameters
   filt : 1-D array or sequence
       Input array.
   trim : str, optional
       A string with 'f' representing trim from front and 'b' to trim from back. Default is 'fb',
       trim zeros from both front and back of the array.

Returns
   trimmed : 1-D array or sequence
       The result of trimming the input. The input data type is preserved.

Examples

>>> a = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
>>> np.trim_zeros(a)
aarray([1, 2, 3, 0, 2, 1])

>>> np.trim_zeros(a, 'b')
aarray([0, 0, 0, 1, 2, 3, 0, 2, 1])

The input data type is preserved, list/tuple in means list/tuple out.
```python
>>> np.trim_zeros([0, 1, 2, 0])
[1, 2]
```

**unique** *(ar, return_index=False, return_inverse=False)*

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.

**Parameters**

- **ar**: array_like
  Input array. This will be flattened if it is not already 1-D.
- **return_index**: bool, optional
  If True, also return the indices of ar that result in the unique array.
- **return_inverse**: bool, optional
  If True, also return the indices of the unique array that can be used to reconstruct ar.

**Returns**

- **unique**: ndarray
  The sorted unique values.
- **unique_indices**: ndarray, optional
  The indices of the unique values in the (flattened) original array. Only provided if return_index is True.
- **unique_inverse**: ndarray, optional
  The indices to reconstruct the (flattened) original array from the unique array. Only provided if return_inverse is True.

**See Also:**

- `numpy.lib.arraysetops`
  Module with a number of other functions for performing set operations on arrays.

**Examples**

```python
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
```

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

Return the indices of the original array that give the unique values:

```python
>>> a = np.array(["a", "b", "b", "c", "a"])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'],
      dtype='|S1')
>>> indices
array([0, 1, 3])
>>> a[indices]
```
array(['a', 'b', 'c'],
      dtype='|S1')

Reconstruct the input array from the unique values:

```python
g = np.array([1, 2, 6, 4, 2, 3, 2])
g, indices = np.unique(g, return_inverse=True)
g
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> g[indices]
g.array([1, 2, 6, 4, 2, 3, 2])
```

### 3.2.9 Rearranging elements

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fliplr(m)</code></td>
<td>Flip array in the left/right direction.</td>
</tr>
<tr>
<td><code>flipud(m)</code></td>
<td>Flip array in the up/down direction.</td>
</tr>
<tr>
<td><code>reshape(a, newshape[, order])</code></td>
<td>Gives a new shape to an array without changing its data.</td>
</tr>
<tr>
<td><code>roll(a, shift[, axis])</code></td>
<td>Roll array elements along a given axis.</td>
</tr>
<tr>
<td><code>rot90(m[, k])</code></td>
<td>Rotate an array by 90 degrees in the counter-clockwise direction.</td>
</tr>
</tbody>
</table>

#### `fliplr(m)`

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Parameters**

- `m` : array_like
  Input array.

**Returns**

- `f` : ndarray
  A view of `m` with the columns reversed. Since a view is returned, this operation is \(O(1)\).

**See Also:**

- `flipud`
  Flip array in the up/down direction.
- `rot90`
  Rotate array counterclockwise.

**Notes**

Equivalent to `A[::,::-1]`. Does not require the array to be two-dimensional.

**Examples**

```python
g = np.diag([1., 2., 3.])
g
array([[1., 0., 0.],
       [0., 2., 0.],
       [0., 0., 3.]])
```
>>> np.fliplr(A)
array([[ 0., 0., 1.],
       [ 0., 2., 0.],
       [ 3., 0., 0.]])

>>> A = np.random.randn(2,3,5)
>>> np.all(numpy.fliplr(A)==A[:,:,::-1])
True

flipud (m)
Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

Parameters

m : array_like
Input array.

Returns

out : array_like
A view of m with the rows reversed. Since a view is returned, this operation is $O(1)$.

See Also:

fliplr
Flip array in the left/right direction.

rot90
Rotate array counterclockwise.

Notes
Equivalent to A[::-1,...]. Does not require the array to be two-dimensional.

Examples

>>> A = np.diag([1.0, 2, 3])

>>> A
array([[ 1., 0., 0.],
       [ 0., 2., 0.],
       [ 0., 0., 3.]])

>>> np.flipud(A)
array([[ 0., 0., 3.],
       [ 0., 2., 0.],
       [ 1., 0., 0.]])

>>> A = np.random.randn(2,3,5)
>>> np.all(np.flipud(A)==A[::-1,...])
True

>>> np.flipud([1,2])
array([2, 1])

reshape (a, newshape, order='C')
Gives a new shape to an array without changing its data.
Parameters

- **a**: array_like
  - Array to be reshaped.

- **newshape**: int or tuple of ints
  - The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

- **order**: {'C', 'F'}, optional
  - Determines whether the array data should be viewed as in C (row-major) order or FORTRAN (column-major) order.

Returns

- **reshaped_array**: ndarray
  - This will be a new view object if possible; otherwise, it will be a copy.

See Also:

**ndarray.reshape**
- Equivalent method.

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
>>> np.reshape(a, (3,-1)) # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

**roll** *(a, shift, axis=None)*

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

Parameters

- **a**: array_like
  - Input array.

- **shift**: int
  - The number of places by which elements are shifted.

- **axis**: int, optional
  - The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

Returns

- **res**: ndarray
  - Output array, with the same shape as `a`.
See Also:

**rollaxis**

Roll the specified axis backwards, until it lies in a given position.

**Examples**

```python
>>> x = np.arange(10)
>>> np.roll(x, 2)
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])

>>> x2 = np.reshape(x, (2,5))
>>> x2
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

>>> np.roll(x2, 1)
array([[9, 0, 1, 2, 3],
       [4, 5, 6, 7, 8]])

>>> np.roll(x2, 1, axis=0)
array([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])

>>> np.roll(x2, 1, axis=1)
array([[4, 0, 1, 2, 3],
       [9, 5, 6, 7, 8]])
```

**rot90** *(m, k=1)*

Rotate an array by 90 degrees in the counter-clockwise direction.

The first two dimensions are rotated; therefore, the array must be at least 2-D.

**Parameters**

- **m**: array_like
  
  Array of two or more dimensions.

- **k**: integer
  
  Number of times the array is rotated by 90 degrees.

**Returns**

- **y**: ndarray
  
  Rotated array.

**See Also:**

**fliplr**

Flip an array horizontally.

**flipud**

Flip an array vertically.

**Examples**

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

```python
>>> m
array([[1, 2],
       [3, 4]])
```

```python
>>> np.rot90(m)
```
array([[2, 4],
       [1, 3]])
>>> np.rot90(m, 2)
array([[4, 3],
       [2, 1]])

3.3 Indexing routines

See Also:
Indexing

3.3.1 Generating index arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c_</code></td>
<td>Translates slice objects to concatenation along the second axis.</td>
</tr>
<tr>
<td><code>r_</code></td>
<td>Translates slice objects to concatenation along the first axis.</td>
</tr>
<tr>
<td><code>s_</code></td>
<td>A nicer way to build up index tuples for arrays.</td>
</tr>
<tr>
<td><code>nonzero</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>where</code></td>
<td>Return elements, either from x or y, depending on condition.</td>
</tr>
<tr>
<td><code>indices</code></td>
<td>Return an array representing the indices of a grid.</td>
</tr>
<tr>
<td><code>ogrid</code></td>
<td>Construct an open mesh from multiple sequences.</td>
</tr>
<tr>
<td><code>unravel_index</code></td>
<td>Convert a flat index to an index tuple for an array of given shape.</td>
</tr>
<tr>
<td><code>diag_indices</code></td>
<td>Return the indices to access the main diagonal of an array.</td>
</tr>
<tr>
<td><code>mask_indices</code></td>
<td>Return the indices to access (n, n) arrays, given a masking function.</td>
</tr>
<tr>
<td><code>tril_indices</code></td>
<td>Return the indices for the lower-triangle of an (n, n) array.</td>
</tr>
<tr>
<td><code>triu_indices</code></td>
<td>Return the indices for the upper-triangle of an (n, n) array.</td>
</tr>
</tbody>
</table>

Examples

```python
>>> np.c_[np.array([[1, 2, 3]]), 0, 0, np.array([[4, 5, 6]])]
array([[1, 2, 3, 0, 0, 4, 5, 6]])
```

```
``` python
>>> np.r_[np.array([[1, 2, 3]]), 0, 0, np.array([[4, 5, 6]])]
```

```
``` python

```python
3.3. Indexing routines 353
```
If slice notation is used, the syntax `start:stop:step` is equivalent to `np.arange(start, stop, step)` inside of the brackets. However, if `step` is an imaginary number (i.e. 100j) then its integer portion is interpreted as a number-of-points desired and the start and stop are inclusive. In other words `start:stop:stepj` is interpreted as `np.linspace(start, stop, step, endpoint=1)` inside of the brackets. After expansion of slice notation, all comma separated sequences are concatenated together.

Optional character strings placed as the first element of the index expression can be used to change the output. The strings ‘r’ or ‘c’ result in matrix output. If the result is 1-D and ‘r’ is specified a 1 x N (row) matrix is produced. If the result is 1-D and ‘c’ is specified, then a N x 1 (column) matrix is produced. If the result is 2-D then both provide the same matrix result.

A string integer specifies which axis to stack multiple comma separated arrays along. A string of two comma-separated integers allows indication of the minimum number of dimensions to force each entry into as the second integer (the axis to concatenate along is still the first integer).

A string with three comma-separated integers allows specification of the axis to concatenate along, the minimum number of dimensions to force the entries to, and which axis should contain the start of the arrays which are less than the specified number of dimensions. In other words the third integer allows you to specify where the 1’s should be placed in the shape of the arrays that have their shapes upgraded. By default, they are placed in the front of the shape tuple. The third argument allows you to specify where the start of the array should be instead. Thus, a third argument of ‘0’ would place the 1’s at the end of the array shape. Negative integers specify where in the new shape tuple the last dimension of upgraded arrays should be placed, so the default is ‘-1’.

### Parameters

- Not a function, so takes no parameters:

### Returns

- A concatenated ndarray or matrix:

### See Also:

- `concatenate`
  - Join a sequence of arrays together.
- `c_`
  - Translates slice objects to concatenation along the second axis.

### Examples

```python
>>> np.r_[np.array([1, 2, 3]), 0, 0, np.array([4, 5, 6])]
array([1, 2, 3, 0, 0, 4, 5, 6])
```

```python
>>> np.r_[-1:1:6j, [0]*3, 5, 6]
array([-1. , -0.6, -0.2, 0.2, 0.6, 1. , 0. , 0. , 0. , 5. , 6. ])
```

String integers specify the axis to concatenate along or the minimum number of dimensions to force entries into.

```python
>>> np.r_['-1', a, a]  # concatenate along last axis
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

```python
>>> np.r_['0', [1,2,3], [4,5,6]]  # concatenate along first axis, dim>=2
array([[1, 2, 3],
       [4, 5, 6]])
```

```python
>>> np.r_['0,2', [1,2,3], [4,5,6]]
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

```python
>>> np.r_['0,2,0', [1,2,3], [4,5,6]]
array([[[1],
        [2],
        [3],
        [4],
        [5],
        [6]],
```
Using ‘r’ or ‘c’ as a first string argument creates a matrix.

```python
>>> np.r_['r',[1,2,3], [4,5,6]]
matrix([[1, 2, 3, 4, 5, 6]])
```

`s_ (maketuple)`

A nicer way to build up index tuples for arrays.

**Note:** Use one of the two predefined instances `index_exp` or `s_` rather than directly using `IndexExpression`.

For any index combination, including slicing and axis insertion, `a[indices]` is the same as `a[np.index_exp[indices]]` for any array `a`. However, `np.index_exp[indices]` can be used anywhere in Python code and returns a tuple of slice objects that can be used in the construction of complex index expressions.

**Parameters**

- `maketuple : bool`
  
  If True, always returns a tuple.

**See Also:**

- `index_exp`
  
  Predefined instance that always returns a tuple: `index_exp = IndexExpression(maketuple=True)`.

- `s_`
  
  Predefined instance without tuple conversion: `s_ = IndexExpression(maketuple=False)`.

**Notes**

You can do all this with `slice()` plus a few special objects, but there’s a lot to remember and this version is simpler because it uses the standard array indexing syntax.

**Examples**

```python
>>> np.s_[:2]
slice(0, 2, None)
>>> np.index_exp[:2]
(slice(0, 2, None),)
```

```python
>>> np.array([0, 1, 2, 3, 4])[np.s_[:2]]
array([0, 1, 2, 3, 4])
```

**nonzero (a)**

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

Returns a tuple of arrays, one for each dimension of `a`, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```python
a[nonzero(a)]
```
To group the indices by element, rather than dimension, use:

```python
transpose(nonzero(a))
```

The result of this is always a 2-D array, with a row for each non-zero element.

**Parameters**

- `a`: array_like
  - Input array.

**Returns**

- `tuple_of_arrays`: tuple
  - Indices of elements that are non-zero.

**See Also:**

- `flatnonzero`
  - Return indices that are non-zero in the flattened version of the input array.

- `ndarray.nonzero`
  - Equivalent ndarray method.

**Examples**

```python
>>> x = np.eye(3)
```

```python
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

```python
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))
```

```python
>>> x[np.nonzero(x)]
array([ 1., 1., 1.])
```

```python
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `np.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```python
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

```python
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the boolean array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```
where()
Return elements, either from x or y, depending on condition.

If only condition is given, return condition.nonzero().

Parameters

condition : array_like, bool
When True, yield x, otherwise yield y.

x, y : array_like, optional
Values from which to choose. x and y need to have the same shape as condition.

Returns

out : ndarray or tuple of ndarrays
If both x and y are specified, the output array contains elements of x where condition is
True, and elements from y elsewhere.
If only condition is given, return the tuple condition.nonzero(), the indices
where condition is True.

See Also:

nonzero, choose

Notes
If x and y are given and input arrays are 1-D, where is equivalent to:

[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]

Examples

>>> np.where([[True, False], [True, True]],
...           [[1, 2], [3, 4]],
...           [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])

>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))

>>> x = np.arange(9.).reshape(3, 3)
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))

>>> x[np.where( x > 3.0 )]     # Note: result is 1D.
array([ 4.,  5.,  6.,  7.,  8.])

>>> np.where(x < 5, x, -1)    # Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])

indices(dimensions, dtype=<type 'int'>)
Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

Parameters

dimensions : sequence of ints

3.3. Indexing routines
The shape of the grid.

**dtype**: dtype, optional

Data type of the result.

Returns

grid : ndarray

The array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

See Also:

`mgrid`, `meshgrid`

Notes

The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if `dimensions` is a tuple `(r0, ..., rN-1)` of length N, the output shape is `(N, r0, ..., rN-1)`.

The subarrays `grid[k]` contains the N-D array of indices along the k-th axis. Explicitly:

\[
grid[k,i0,i1,...,iN-1] = ik
\]

Examples

```python
>>> grid = np.indices((2, 3))
>>> grid.shape
(2,2,3)
>>> grid[0]  # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]  # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```python
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```

Note that it would be more straightforward in the above example to extract the required elements directly with `x[:2, :3]`.

**ix_\(*args\)**

Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions.

Using `ix_` one can quickly construct index arrays that will index the cross product. `a[np.ix_([1,3],[2,5])]` returns the array `[[a[1,2] a[1,5]], [a[3,2] a[3,5]]].`

Parameters

- **args**: 1-D sequences
Returns

out : tuple of ndarrays

N arrays with N dimensions each, with N the number of input sequences. Together these arrays form an open mesh.

See Also:

ogrid, mgrid, meshgrid

Examples

```python
>>> a = np.arange(10).reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> ixgrid = np.ix_([0, 1], [2, 4])
>>> ixgrid
(array([[0],
        [1]]), array([[2, 4]]))
>>> ixgrid[0].shape, ixgrid[1].shape
((2, 1), (1, 2))
>>> a[ixgrid]
array([[2, 4],
       [7, 9]])
```

ogrid(sparse=False)

nd_grid instance which returns an open multi-dimensional “meshgrid”.

An instance of numpy.lib.index_tricks.nd_grid which returns an open (i.e. not fleshed out) meshgrid when indexed, so that only one dimension of each returned array is greater than 1. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a complex number (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

Returns

mesh-grid ‘ndarrays’ with only one dimension \( \text{neq 1} \):

See Also:

np.lib.index_tricks.nd_grid
class of ogrid and mgrid objects

mgrid

like ogrid but returns dense (or fleshed out) mesh grids

r_

array concatenator

Examples

```python
>>> from numpy import ogrid
>>> ogrid[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
>>> ogrid[0:5,0:5]
[array([[0],
       [1],
       [2],
       [3],
       [4]], dtype=int64),
array([[0, 1, 2, 3, 4]])]
```
unravel_index(x, dims)
Convert a flat index to an index tuple for an array of given shape.

Parameters
x : int
    Flattened index.
dims : tuple of ints
    Input shape, the shape of an array into which indexing is required.

Returns
idx : tuple of ints
    Tuple of the same shape as dims, containing the unraveled index.

Notes
In the Examples section, since arr.flat[x] == arr.max() it may be easier to use flattened indexing
than to re-map the index to a tuple.

Examples
>>> arr = np.arange(20).reshape(5, 4)
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> x = arr.argmax()
>>> x
19
>>> dims = arr.shape
>>> idx = np.unravel_index(x, dims)
>>> idx
(4, 3)
>>> arr[idx] == arr.max()
True

diag_indices(n, ndim=2)
Return the indices to access the main diagonal of an array.

This returns a tuple of indices that can be used to access the main diagonal of an array a with a.ndim >= 2
dimensions and shape (n, n, ..., n). For a.ndim = 2 this is the usual diagonal, for a.ndim > 2 this is the
set of indices to access a[i, i, ..., i] for i = [0..n-1].

Parameters
n : int
    The size, along each dimension, of the arrays for which the returned indices can be used.
ndim : int, optional
    The number of dimensions.
See Also:

diag_indices_from

Notes
New in version 1.4.0.

Examples
Create a set of indices to access the diagonal of a (4, 4) array:

```python
>>> di = np.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))

>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

>>> a[di] = 100
>>> a
array([[100,  1,  2,  3],
       [ 4, 100,  6,  7],
       [ 8,  9, 100, 11],
       [12, 13, 14, 100]])
```

Now, we create indices to manipulate a 3-D array:

```python
>>> d3 = np.diag_indices(2, 3)
>>> d3
(array([0, 1]), array([0, 1]), array([0, 1]))

And use it to set the diagonal of an array of zeros to 1:

```python
>>> a = np.zeros((2, 2, 2), dtype=np.int)
>>> a[d3] = 1
>>> a
array([[[1, 0],
         [0, 0]],
       [[0, 0],
         [0, 1]]])
```

diag_indices_from(arr)

Return the indices to access the main diagonal of an n-dimensional array.

See diag_indices for full details.

Parameters

- **arr**: array, at least 2-D

See Also:

diag_indices

Notes
New in version 1.4.0.
**mask_indices** *(n, mask_func, k=0)*

Return the indices to access (n, n) arrays, given a masking function.

Assume `mask_func` is a function that, for a square array `a` of size `(n, n)` with a possible offset argument `k`, when called as `mask_func(a, k)` returns a new array with zeros in certain locations (functions like `triu` or `tril` do precisely this). Then this function returns the indices where the non-zero values would be located.

**Parameters**

- **n**: int
  - The returned indices will be valid to access arrays of shape (n, n).

- **mask_func**: callable
  - A function whose call signature is similar to that of `triu`, `tril`. That is, `mask_func(x, k)` returns a boolean array, shaped like `x`. `k` is an optional argument to the function.

- **k**: scalar
  - An optional argument which is passed through to `mask_func`. Functions like `triu`, `tril` take a second argument that is interpreted as an offset.

**Returns**

- **indices**: tuple of arrays.
  - The `n` arrays of indices corresponding to the locations where `mask_func(np.ones((n, n)), k)` is True.

**See Also:**

`triu`, `tril`, `triu_indices`, `tril_indices`

**Notes**

New in version 1.4.0.

**Examples**

These are the indices that would allow you to access the upper triangular part of any 3x3 array:

```python
>>> iu = np.mask_indices(3, np.triu)
```

For example, if `a` is a 3x3 array:

```python
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a[iu]
array([0, 1, 2, 4, 5, 8])
```

An offset can be passed also to the masking function. This gets us the indices starting on the first diagonal right of the main one:

```python
>>> iul = np.mask_indices(3, np.triu, 1)
```

with which we now extract only three elements:

```python
>>> a[iul]
array([1, 2, 5])
```
tril_indices \((n, k=0)\)

Return the indices for the lower-triangle of an \((n, n)\) array.

Parameters

- **n** : int
  
  Sets the size of the arrays for which the returned indices will be valid.

- **k** : int, optional
  
  Diagonal offset (see tril for details).

Returns

- **inds** : tuple of arrays
  
  The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array.

See Also:

- triu_indices
  
  similar function, for upper-triangular.

- mask_indices
  
  generic function accepting an arbitrary mask function.

tril, triu

Notes

New in version 1.4.0.

Examples

Compute two different sets of indices to access 4x4 arrays, one for the lower triangular part starting at the main diagonal, and one starting two diagonals further right:

```python
>>> il1 = np.tril_indices(4)
>>> il2 = np.tril_indices(4, 2)
```

Here is how they can be used with a sample array:

```python
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Both for indexing:

```python
>>> a[il1]
array([ 0,  4,  5,  8,  9, 10, 12, 13, 14, 15])
```

And for assigning values:

```python
>>> a[il1] = -1
>>> a
array([[-1,  1,  2,  3],
       [-1, -1,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```
[-1, -1, -1, 11],
[-1, -1, -1, -1])

These cover almost the whole array (two diagonals right of the main one):

```python
>>> a[il2] = -10
```
```python
>>> a
array([[-10, -10, -10,  3],
        [-10, -10, -10, -10],
        [-10, -10, -10, -10],
        [-10, -10, -10, -10]])
```

**tril_indices_from** *(arr, k=0)*

Return the indices for the lower-triangle of an (n, n) array.

See :meth:`tril_indices` for full details.

**Parameters**

- **n** : int
  
  Sets the size of the arrays for which the returned indices will be valid.

- **k** : int, optional
  
  Diagonal offset (see :meth:`tril` for details).

**See Also:**

:func:`tril_indices`, :func:`tril`

**Notes**

New in version 1.4.0.

**triu_indices** *(n, k=0)*

Return the indices for the upper-triangle of an (n, n) array.

**Parameters**

- **n** : int
  
  Sets the size of the arrays for which the returned indices will be valid.

- **k** : int, optional
  
  Diagonal offset (see :func:`triu` for details).

**Returns**

- **inds** : tuple of arrays
  
  The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array.

**See Also:**

:func:`tril_indices`

Similar function, for lower-triangular.

:func:`mask_indices`

Generic function accepting an arbitrary mask function.

:func:`triu`, :func:`tril`
Notes

New in version 1.4.0.

Examples

Compute two different sets of indices to access 4x4 arrays, one for the upper triangular part starting at the main diagonal, and one starting two diagonals further right:

```python
>>> iu1 = np.triu_indices(4)
>>> iu2 = np.triu_indices(4, 2)
```

Here is how they can be used with a sample array:

```python
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Both for indexing:

```python
>>> a[iu1]
array([ 0,  1,  2,  3,  5,  6,  7, 10, 11, 15])
```

And for assigning values:

```python
>>> a[iu1] = -1
>>> a
array([[ -1, -1, -10, -10],
       [ 4, -1, -1, -10],
       [ 8,  9, -1, -10],
       [12, 13, 14, -1]])
```

These cover only a small part of the whole array (two diagonals right of the main one):

```python
>>> a[iu2] = -10
>>> a
array([[ -1, -1, -10, -10],
       [ 4, -1, -1, -10],
       [ 8,  9, -1, -10],
       [12, 13, 14, -1]])
```

`triu_indices_from(arr, k=0)`

Return the indices for the lower-triangle of an (n, n) array.

See `triu_indices` for full details.

**Parameters**

- **n** : int
  
  Sets the size of the arrays for which the returned indices will be valid.

- **k** : int, optional
  
  Diagonal offset (see `triu` for details).
See Also:

triu_indices, triu

Notes

New in version 1.4.0.

### 3.3.2 Indexing-like operations

**take** *(a, indices[, axis, out, mode]*)

Take elements from an array along an axis.

This function does the same thing as “fancy” indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis.

**Parameters**

- **a**: array_like
  - The source array.
- **indices**: array_like
  - The indices of the values to extract.
- **axis**: int, optional
  - The axis over which to select values. By default, the flattened input array is used.
- **out**: ndarray, optional
  - If provided, the result will be placed in this array. It should be of the appropriate shape and dtype.
- **mode**: {‘raise’, ‘wrap’, ‘clip’}, optional
  - Specifies how out-of-bounds indices will behave.
  - ‘raise’ – raise an error (default)
  - ‘wrap’ – wrap around
  - ‘clip’ – clip to the range

‘clip’ mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

**Returns**

- **subarray**: ndarray
  - The returned array has the same type as a.

**See Also:**

ndarray.take

equivalent method
Examples

```python
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> np.take(a, indices)
array([4, 3, 6])
```

In this example if `a` is an ndarray, “fancy” indexing can be used.

```python
>>> a = np.array(a)
>>> a[indices]
array([4, 3, 6])
```

`choose`(a, choices, out=None, mode='raise')

Construct an array from an index array and a set of arrays to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = numpy.lib.index_tricks):

```python
np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)]).
```

But this omits some subtleties. Here is a fully general summary:

Given an “index” array (`a`) of integers and a sequence of `n` arrays (`choices`), `a` and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these `Ba` and `Bchoices[i]`, `i = 0,...,n-1` we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each `i`. Then, a new array with shape `Ba.shape` is created as follows:

- if `mode=raise` (the default), then, first of all, each element of `a` (and thus `Ba`) must be in the range `[0, n-1]`; now, suppose that `i` (in that range) is the value at the `(j0, j1, ...., jm)` position in `Ba` - then the value at the same position in the new array is the value in `Bchoices[i]` at that same position;
- if `mode=wrap`, values in `a` (and thus `Ba`) may be any (signed) integer; modular arithmetic is used to map integers outside the range `[0, n-1]` back into that range; and then the new array is constructed as above;
- if `mode=clip`, values in `a` (and thus `Ba`) may be any (signed) integer; negative integers are mapped to 0; values greater than `n-1` are mapped to `n-1`; and then the new array is constructed as above.

Parameters

- `a` : int array
  This array must contain integers in `[0, n-1]`, where `n` is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.

- `choices` : sequence of arrays
  Choice arrays. `a` and all of the choices must be broadcastable to the same shape. If `choices` is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the “sequence”.

- `out` : array, optional
  If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

- `mode` : {'raise' (default), 'wrap', 'clip'}, optional
  Specifies how indices outside `[0, n-1]` will be treated:
  - ‘raise’ : an exception is raised
  - ‘wrap’ : value becomes value mod `n`
**NumPy Reference, Release 1.4.1**

- `clip` : values < 0 are mapped to 0, values > n-1 are mapped to n-1

**Returns**

- merged_array : array
  
  The merged result.

**Raises**

- ValueError: shape mismatch :
  
  If a and each choice array are not all broadcastable to the same shape.

**See Also**

- ndarray.choose
  
  equivalent method

**Notes**

To reduce the chance of misinterpretation, even though the following “abuse” is nominally supported, choices should neither be, nor be thought of as, a single array, i.e., the outermost sequence-like container should be either a list or a tuple.

**Examples**

```python
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
   ... [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices)
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... array([20, 31, 12, 3])
>>> np.choose([2, 4, 1, 0], choices, mode='clip')
... # 4 goes to 3 (4-1)
array([20, 31, 12, 3])
>>> # because there are 4 choice arrays
>>> np.choose([2, 4, 1, 0], choices, mode='wrap')
... # 4 goes to (4 mod 4)
array([20, 1, 12, 3])
>>> # i.e., 0

A couple examples illustrating how choose broadcasts:

```python
>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> np.choose(a, choices)
array([[ 10, -10, 10],
       [-10, 10, -10],
       [ 10, -10, 10]])
```
compress (condition, a, axis=None, out=None)

Return selected slices of an array along given axis.

When working along a given axis, a slice along that axis is returned in output for each index where condition evaluates to True. When working on a 1-D array, compress is equivalent to extract.

Parameters:
- condition : 1-D array of bools
  Array that selects which entries to return. If len(condition) is less than the size of a along the given axis, then output is truncated to the length of the condition array.
- a : array_like
  Array from which to extract a part.
- axis : int, optional
  Axis along which to take slices. If None (default), work on the flattened array.
- out : ndarray, optional
  Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns:
- compressed_array : ndarray
  A copy of a without the slices along axis for which condition is false.

See Also:
take, choose, diag, diagonal, select
ndarray.compress
Equivalent method.
numpy.doc.ufuncs
Section “Output arguments”

Examples

```python
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.compress([0, 1], a, axis=0)
array([[1, 2],
       [3, 4]])
>>> np.compress([False, True, True], a, axis=0)
array([[3, 4],
       [5, 6]])
>>> np.compress([False, True], a, axis=1)
array([[2],
       [4],
       [6]])
```

Working on the flattened array does not return slices along an axis but selects elements.
```python
>>> np.compress([False, True], a)
array([2])
```

`diag(v, k=0)`

Extract a diagonal or construct a diagonal array.

**Parameters**
- `v` : array_like
  - If `v` is a 2-D array, return a copy of its `k`-th diagonal. If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th diagonal.
- `k` : int, optional
  - Diagonal in question. The default is 0. Use `k>0` for diagonals above the main diagonal, and `k` for diagonals below the main diagonal.

**Returns**
- `out` : ndarray
  - The extracted diagonal or constructed diagonal array.

**See Also:**
- `diagonal`
  - Return specified diagonals.
- `diagflat`
  - Create a 2-D array with the flattened input as a diagonal.
- `trace`
  - Sum along diagonals.
- `triu`
  - Upper triangle of an array.
- `tril`
  - Lower triangle of an array.

**Examples**
```python
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.diag(x)
array([0, 4, 8])

>>> np.diag(x, k=1)
array([1, 5])

>>> np.diag(x, k=-1)
array([3, 7])

>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```
diagonal (a, offset=0, axis1=0, axis2=1)
Return specified diagonals.
If \(a\) is 2-D, returns the diagonal of \(a\) with the given offset, i.e., the collection of elements of the form \(a[i, i+offset]\).
If \(a\) has more than two dimensions, then the axes specified by \(axis1\) and \(axis2\) are used to determine the 2-D subarray whose diagonal is returned. The shape of the resulting array can be determined by removing \(axis1\) and \(axis2\) and appending an index to the right equal to the size of the resulting diagonals.

Parameters
- **a**: array_like
  Array from which the diagonals are taken.
- **offset**: int, optional
  Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to main diagonal (0).
- **axis1**: int, optional
  Axis to be used as the first axis of the 2-D subarrays from which the diagonals should be taken. Defaults to first axis (0).
- **axis2**: int, optional
  Axis to be used as the second axis of the 2-D subarrays from which the diagonals should be taken. Defaults to second axis (1).

Returns
- **array_of_diagonals**: ndarray
  If \(a\) is 2-D, a 1-D array containing the diagonal is returned. If \(a\) has larger dimensions, then an array of diagonals is returned.

Raises
- **ValueError**: If the dimension of \(a\) is less than 2.

See Also:
- **diag**: Matlab workalike for 1-D and 2-D arrays.
- **diagflat**: Create diagonal arrays.
- **trace**: Sum along diagonals.

Examples

```python
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])
```
```python
>>> a = np.arange(8).reshape(2,2,2)
>>> a
array([[[0, 1],
        [2, 3]],
        [[4, 5],
        [6, 7]]])
>>> a.diagonal(0,-2,-1)
array([[0, 3],
       [4, 7]])
```

**select** (*condlist*, *choicelist*, *default=0*)

Return an array drawn from elements in choicelist, depending on conditions.

**Parameters**

- **condlist** : list of bool ndarrays
  
The list of conditions which determine from which array in *choicelist* the output elements are taken. When multiple conditions are satisfied, the first one encountered in *condlist* is used.
  
- **choicelist** : list of ndarrays
  
The list of arrays from which the output elements are taken. It has to be of the same length as *condlist*.
  
- **default** : scalar, optional
  
The element inserted in *output* when all conditions evaluate to False.

**Returns**

- **output** : ndarray
  
The output at position m is the m-th element of the array in *choicelist* where the m-th element of the corresponding array in *condlist* is True.

**See Also:**

- **where**
  
Return elements from one of two arrays depending on condition.

- **take**, **choose**, **compress**, **diag**, **diagonal**

**Examples**

```python
>>> x = np.arange(10)
>>> condlist = [x<3, x>5]
>>> choicelist = [x, x**2]
>>> np.select(condlist, choicelist)
array([ 0,  1,  2,  0,  0, 36, 49, 64, 81])
```

### 3.3.3 Inserting data into arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>place()</code></td>
<td>Change elements of an array based on conditional and input values.</td>
</tr>
<tr>
<td><code>put()</code></td>
<td>Replaces specified elements of an array with given values.</td>
</tr>
<tr>
<td><code>putmask()</code></td>
<td>Changes elements of an array based on conditional and input values.</td>
</tr>
<tr>
<td><code>fill_diagonal()</code></td>
<td>Fill the main diagonal of the given array of any dimensionality.</td>
</tr>
</tbody>
</table>

---
place(arr, mask, vals)

Change elements of an array based on conditional and input values.

Similar to np.putmask(a, mask, vals), the difference is that place uses the first N elements of vals, where N is the number of True values in mask, while putmask uses the elements where mask is True.

Note that extract does the exact opposite of place.

Parameters

a : array_like
    Array to put data into.

mask : array_like
    Boolean mask array. Must have the same size as a.

vals : 1-D sequence
    Values to put into a. Only the first N elements are used, where N is the number of True values in mask. If vals is smaller than N it will be repeated.

See Also:

putmask, put, take, extract

Examples

```python
>>> x = np.arange(6).reshape(2, 3)
>>> np.place(x, x>2, [44, 55])
>>> x
array([[ 0,  1,  2],
       [44, 55, 44]])
```

put(a, ind, v, mode='raise')

Replaces specified elements of an array with given values.

The indexing works on the flattened target array. put is roughly equivalent to:

```python
a.flat[ind] = v
```

Parameters

a : ndarray
    Target array.

ind : array_like
    Target indices, interpreted as integers.

v : array_like
    Values to place in a at target indices. If v is shorter than ind it will be repeated as necessary.

mode : {'raise', 'wrap', 'clip'}, optional
    Specifies how out-of-bounds indices will behave.

    • 'raise' – raise an error (default)
    • 'wrap' – wrap around
    • 'clip' – clip to the range
‘clip’ mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

See Also:
putmask, place

Examples

```python
>>> a = np.arange(5)
>>> np.put(a, [0, 2], [-44, -55])
array([-44,  1, -55,  3,  4])

>>> a = np.arange(5)
>>> np.put(a, 22, -5, mode='clip')
array([ 0,  1,  2,  3, -5])
```

putmask()
Changes elements of an array based on conditional and input values.
If values is not the same size as a and mask then it will repeat. This gives behavior different from a[mask] = values.

Parameters
- a : array_like
  Target array.
- mask : array_like
  Boolean mask array. It has to be the same shape as a.
- values : array_like
  Values to put into a where mask is True. If values is smaller than a it will be repeated.

See Also:
place, put, take

Examples

```python
>>> x = np.arange(6).reshape(2, 3)
>>> np.putmask(x, x>2, x**2)
>>> x
array([[  0,   1,   2],
        [  9, 16, 25]])
```

If values is smaller than a it is repeated:

```python
>>> x = np.arange(5)
>>> np.putmask(x, x>1, [-33, -44])
>>> x
array([  0,  1, -33, -44, -33])
```
fill_diagonal \((a, \text{val})\)

Fill the main diagonal of the given array of any dimensionality.

For an array \(a\) with \(a.\text{ndim} > 2\), the diagonal is the list of locations with indices \(a[i, i, \ldots, i]\) all identical. This function modifies the input array in-place, it does not return a value.

**Parameters**

- **a**: array, at least 2-D.
  
  Array whose diagonal is to be filled, it gets modified in-place.

- **val**: scalar
  
  Value to be written on the diagonal, its type must be compatible with that of the array \(a\).

**See Also:**

diag_indices, diag_indices_from

**Notes**

New in version 1.4.0. This functionality can be obtained via diag_indices, but internally this version uses a much faster implementation that never constructs the indices and uses simple slicing.

**Examples**

```python
>>> a = zeros((3, 3), int)
>>> fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])
```

The same function can operate on a 4-D array:

```python
>>> a = zeros((3, 3, 3, 3), int)
>>> fill_diagonal(a, 4)
```

We only show a few blocks for clarity:

```python
>>> a[0, 0]
array([[4, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> a[1, 1]
array([[0, 0, 0],
       [4, 0, 0],
       [0, 0, 0]])
>>> a[2, 2]
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 4]])
```

### 3.3.4 Iterating over arrays

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndenumerate</td>
</tr>
<tr>
<td>ndindex</td>
</tr>
<tr>
<td>flatiter</td>
</tr>
</tbody>
</table>

3.3. Indexing routines
class `ndenumerate`(arr)

Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

Parameters

- `a` : ndarray
  Input array.

See Also:

`ndindex`, `flatiter`

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> for index, x in np.ndenumerate(a):
...     print(index, x)
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

Methods

class `ndindex`(*args)

An N-dimensional iterator object to index arrays.

Given the shape of an array, an `ndindex` instance iterates over the N-dimensional index of the array. At each iteration a tuple of indices is returned, the last dimension is iterated over first.

Parameters

- `*args` : ints
  The size of each dimension of the array.

See Also:

`ndenumerate`, `flatiter`

Examples

```python
>>> for index in np.ndindex(3, 2, 1):
...     print(index)
(0, 0, 0)
(0, 1, 0)
(1, 0, 0)
(1, 1, 0)
(2, 0, 0)
(2, 1, 0)
```

Methods

class `flatiter`()

Flat iterator object to iterate over arrays.
A `flatiter` iterator is returned by `x.flat` for any array `x`. It allows iterating over the array as if it were a 1-D array, either in a for-loop or by calling its `next` method.

Iteration is done in C-contiguous style, with the last index varying the fastest. The iterator can also be indexed using basic slicing or advanced indexing.

**See Also:**

- `ndarray.flat`
  Return a flat iterator over an array.
- `ndarray.flatten`
  Returns a flattened copy of an array.

**Notes**

A `flatiter` iterator can not be constructed directly from Python code by calling the `flatiter` constructor.

**Examples**

```python
>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> type(fl)
<type 'numpy.flatiter'>
>>> for item in fl:
...     print(item)
...    0
...      1
...        2
...    3
...      4
...    5
```

```python
>>> fl[2:4]
array([2, 3])
```

**Methods**

### 3.4 Data type routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>can_cast(fromtype, totype)</code></td>
<td>Returns True if cast between data types can occur without losing precision.</td>
</tr>
<tr>
<td><code>common_type(*arrays)</code></td>
<td>Return a scalar type which is common to the input arrays.</td>
</tr>
<tr>
<td><code>obj2sctype(rep[, default])</code></td>
<td></td>
</tr>
</tbody>
</table>

**`can_cast()`**

Returns True if cast between data types can occur without losing precision.

**Parameters**

- **fromtype** : dtype or dtype specifier
  Data type to cast from.

- **totype** : dtype or dtype specifier
  Data type to cast to.
Returns

```
out : bool

True if cast can occur without losing precision.
```

Examples

```
>>> np.can_cast(np.int32, np.int64)
True
>>> np.can_cast(np.float64, np.complex)
True
>>> np.can_cast(np.complex, np.float)
False

>>> np.can_cast('i8', 'f8')
True
>>> np.can_cast('i8', 'f4')
False
>>> np.can_cast('i4', 'S4')
True
```

`common_type(*arrays)`

Return a scalar type which is common to the input arrays.

The return type will always be an inexact (i.e. floating point) scalar type, even if all the arrays are integer arrays. If one of the inputs is an integer array, the minimum precision type that is returned is a 64-bit floating point dtype.

All input arrays can be safely cast to the returned dtype without loss of information.

Parameters

```
array1, array2, ... : ndarrays

Input arrays.
```

Returns

```
out : data type code

Data type code.
```

See Also:

`dtype`, `mintypecode`

Examples

```
>>> np.common_type(np.arange(2, dtype=np.float32))
<type 'numpy.float32'>
>>> np.common_type(np.arange(2, dtype=np.float32), np.arange(2))
<type 'numpy.float64'>
>>> np.common_type(np.arange(4), np.array([45, 6.j]), np.array([45.0]))
<type 'numpy.complex128'>
```

`obj2sctype(rep, default=None)`
3.4.1 Creating data types

```
dtype (obj[, align, copy])  # Create a data type object.

format_parser  # Class to convert formats, names, titles description to a dtype.
```

class `dtype` ()

Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

**Parameters**

- `obj`:
  
  Object to be converted to a data type object.

- `align` (bool, optional)
  
  Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be `True` only if `obj` is a dictionary or a comma-separated string.

- `copy` (bool, optional)
  
  Make a new copy of the data-type object. If `False`, the result may just be a reference to a built-in data-type object.

**Examples**

Using array-scalar type:

```python
>>> np.dtype(np.int16)
dtype('int16')
```

Record, one field name ‘f1’, containing int16:

```python
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Record, one field named ‘f1’, in itself containing a record with one field:

```python
>>> np.dtype([('f1', np.int16)])
dtype([('f1', [('f1', '<i2')])])
```

Record, two fields: the first field contains an unsigned int, the second an int32:

```python
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```

Using array-protocol type strings:

```python
>>> np.dtype([('a', 'f8'), ('b', 'S10')])
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```python
>>> np.dtype("i4, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

Using tuples. `int` is a fixed type, 3 the field’s shape. `void` is a flexible type, here of size 10:
```python
>>> np.dtype([(‘hello’, (np.int, 3)), (‘world’, np.void, 10))
dtype([(‘hello’, ‘<i4’, 3), (‘world’, ‘|V10’)])
```

Subdivide `int16` into 2 `int8`'s, called `x` and `y`. 0 and 1 are the offsets in bytes:

```python
>>> np.dtype((np.int16, {‘x’:(np.int8, 0), ‘y’:(np.int8, 1)}))
dtype({‘<i2’, [{‘x’, ‘|i1’}, {‘y’, ‘|i1’}])
```

Using dictionaries. Two fields named ‘gender’ and ‘age’:

```python
>>> np.dtype({‘names’:[‘gender’,’age’], ‘formats’:[‘S1’,np.uint8]})
dtype({‘gender’, ‘|S1’}, {‘age’, ‘|u1’})
```

Offsets in bytes, here 0 and 25:

```python
>>> np.dtype({‘surname’:{‘S25’, 0},’age’:(np.uint8, 25)})
dtype({‘surname’, ‘|S25’}, {‘age’, ‘|u1’})
```

### Methods

**class format_parser**( formats, names, titles, aligned=False, byteorder=None)

Class to convert formats, names, titles description to a dtype.

After constructing the format_parser object, the dtype attribute is the converted data-type: **dtype** = format_parser(formats, names, titles).dtype

**Parameters**

- **formats**: str or list of str
  The format description, either specified as a string with comma-separated format descriptions in the form ‘f8, i4, a5’, or a list of format description strings in the form [‘f8’, ‘i4’, ‘a5’].

- **names**: str or list/tuple of str
  The field names, either specified as a comma-separated string in the form ‘col1, col2, col3’, or as a list or tuple of strings in the form [‘col1’, ‘col2’, ‘col3’]. An empty list can be used, in that case default field names (‘f0’, ‘f1’, ...) are used.

- **titles**: sequence
  Sequence of title strings. An empty list can be used to leave titles out.

- **aligned**: bool, optional
  If True, align the fields by padding as the C-compiler would. Default is False.

- **byteorder**: str, optional
  If specified, all the fields will be changed to the provided byte-order. Otherwise, the default byte-order is used. For all available string specifiers, see `dtype.newbyteorder`.

**See Also**:

dtype, typename, sctype2char
Examples

```python
>>> np.format_parser(['f8', 'i4', 'a5'], ['col1', 'col2', 'col3'], ...)['T1', 'T2', 'T3']].dtype
dtype([('T1', 'col1'), '<f8'), ('T2', 'col2'), '<i4'),
     ('T3', 'col3'), '|S5')]

names and/or titles can be empty lists. If titles is an empty list, titles will simply not appear. If names is empty,
default field names will be used.

```python
>>> np.format_parser(['f8', 'i4', 'a5'], ['col1', 'col2', 'col3'], ...)[]].dtype
dtype([('col1', '<f8'), ('col2', '<i4'), ('col3', '|S5')])

```python
>>> np.format_parser(['f8', 'i4', 'a5'], [], []).dtype
dtype([('f0', '<f8'), ('f1', '<i4'), ('f2', '|S5')])
```

Attributes

<table>
<thead>
<tr>
<th>dtype</th>
<th>dtype</th>
<th>The converted data-type.</th>
</tr>
</thead>
</table>

3.4.2 Data type information

<table>
<thead>
<tr>
<th>finfo (dtype)</th>
<th>Machine limits for floating point types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>iinfo (type)</td>
<td>Machine limits for integer types.</td>
</tr>
<tr>
<td>MachAr</td>
<td>Diagnosing machine parameters.</td>
</tr>
</tbody>
</table>

class `finfo`()

Machine limits for floating point types.

Parameters

- `dtype` : floating point type, dtype, or instance.
  The kind of floating point data type to get information about.

See Also:

- `MachAr`
  The implementation of the tests that produce this information.

- `iinfo`
  The equivalent for integer data types.

Notes

For developers of NumPy: do not instantiate this at the module level. The initial calculation of these parameters
is expensive and negatively impacts import times. These objects are cached, so calling `finfo()` repeatedly
inside your functions is not a problem.
### Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eps</td>
<td>Floating point number of the appropriate type</td>
</tr>
<tr>
<td>epsneg</td>
<td>Floating point number of the appropriate type</td>
</tr>
<tr>
<td>iexp</td>
<td>Integer</td>
</tr>
<tr>
<td>machar</td>
<td>MachAr</td>
</tr>
<tr>
<td>machep</td>
<td>Integer</td>
</tr>
<tr>
<td>max</td>
<td>Floating point number of the appropriate type</td>
</tr>
<tr>
<td>maxexp</td>
<td>Integer</td>
</tr>
<tr>
<td>min</td>
<td>Floating point number of the appropriate type</td>
</tr>
<tr>
<td>minexp</td>
<td>Integer</td>
</tr>
<tr>
<td>negep</td>
<td>Integer</td>
</tr>
<tr>
<td>nexp</td>
<td>Integer</td>
</tr>
<tr>
<td>nmant</td>
<td>Integer</td>
</tr>
<tr>
<td>precision</td>
<td>Floating point number of the appropriate type</td>
</tr>
<tr>
<td>resolution</td>
<td>Floating point number of the appropriate type</td>
</tr>
<tr>
<td>tiny</td>
<td>Floating point number of the appropriate type</td>
</tr>
</tbody>
</table>
| **class**`iinfo(type)` **Machine limits for integer types.**
- **Parameters**
  - `type`: integer type, dtype, or instance
    - The kind of integer data type to get information about.
- **See Also**:
  - `finfo`
    - The equivalent for floating point data types.

### Examples

With types:

```python
>>> ii16 = np.iinfo(np.int16)
>>> ii16.min
-32768
>>> ii16.max
32767
>>> ii32 = np.iinfo(np.int32)
>>> ii32.min
-2147483648
>>> ii32.max
2147483647
```

With instances:
>>> ii32 = np.iinfo(np.int32(10))
>>> ii32.min
-2147483648
>>> ii32.max
2147483647

Attributes

class MachAr (float_conv=<type 'float'>, int_conv=<type 'int'>, float_to_float=<type 'float'>,
 float_to_str=<function <lambda> at 0x1e77c70>, title='Python floating point number')

Diagnosing machine parameters.

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

See Also:

finfo
  Machine limits for floating point types.
iinfo
  Machine limits for integer types.

References

[R1]

3.4. Data type routines 383
## Attributes

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

### 3.4.3 Data type testing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{issctype}(rep)</td>
<td>Determines whether the given object represents a scalar data-type.</td>
</tr>
<tr>
<td>\texttt{issubdtype}(arg1, arg2)</td>
<td>Returns True if first argument is a typecode lower/equal in type hierarchy.</td>
</tr>
<tr>
<td>\texttt{issubsctype}(arg1, arg2)</td>
<td>Determine if the first argument is a subclass of the second argument.</td>
</tr>
<tr>
<td>\texttt{issubclass_}(arg1, arg2)</td>
<td>Determine common type following standard coercion rules.</td>
</tr>
<tr>
<td>\texttt{find_common_type}(array_types, scalar_types)</td>
<td></td>
</tr>
</tbody>
</table>

**issctype**(rep)

Determines whether the given object represents a scalar data-type.

**Parameters**

- **rep** : any

  If \(rep\) is an instance of a scalar dtype, True is returned. If not, False is returned.

**Returns**

- **out** : bool

  Boolean result of check whether \(rep\) is a scalar dtype.

**See Also:**

\texttt{issubsctype}, \texttt{issubdtype}, \texttt{obj2sctype}, \texttt{sctype2char}
Examples

>>> np.issctype(np.int32)
True
>>> np.issctype(list)
False
>>> np.issctype(1.1)
False

issubdtype (arg1, arg2)
Returns True if first argument is a typecode lower/equal in type hierarchy.

Parameters
arg1, arg2 : dtype_like
dtype or string representing a typecode.

Returns
out : bool

See Also:
issubsctype, issubclass_

numpy.core.numerictypes
Overview of numpy type hierarchy.

Examples

>>> np.issubdtype(‘S1’, str)
True
>>> np.issubdtype(np.float64, np.float32)
False

issubsctype (arg1, arg2)
Determine if the first argument is a subclass of the second argument.

Parameters
arg1, arg2 : dtype or dtype specifier
Data-types.

Returns
out : bool
The result.

See Also:
issctype, issubdtype, obj2sctype

Examples

>>> np.issubsctype(‘S8’, str)
True
>>> np.issubsctype(np.array([1]), np.int)
True
>>> np.issubsctype(np.array([1]), np.float)
False

issubclass_ (arg1, arg2)
find_common_type (array_types, scalar_types)

Determine common type following standard coercion rules.

Parameters

array_types : sequence
    A list of dtypes or dtype convertible objects representing arrays.
scalar_types : sequence
    A list of dtypes or dtype convertible objects representing scalars.

Returns

datatype : dttype
    The common data type, which is the maximum of array_types ignoring scalar_types, unless the maximum of scalar_types is of a different kind (dtype.kind). If the kind is not understood, then None is returned.

See Also:
dtype, common_type, can_cast, mintypecode

Examples

>>> np.find_common_type([], [np.int64, np.float32, np.complex])
dtype('complex128')
>>> np.find_common_type([np.int64, np.float32], [])
dtype('float64')

The standard casting rules ensure that a scalar cannot up-cast an array unless the scalar is of a fundamentally different kind of data (i.e. under a different hierarchy in the data type hierarchy) then the array:

>>> np.find_common_type([np.float32], [np.int64, np.float64])
dtype('float32')

Complex is of a different type, so it up-casts the float in the array_types argument:

>>> np.find_common_type([np.float32], [np.complex])
dtype('complex128')

Type specifier strings are convertible to dtypes and can therefore be used instead of dtypes:

>>> np.find_common_type(['f4', 'f4', 'i4'], ['c8'])
dtype('complex128')

3.4.4 Miscellaneous

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typename (char)</td>
<td>Return a description for the given data type code.</td>
</tr>
<tr>
<td>sctype2char (sctype)</td>
<td>Return the string representation of a scalar dtype.</td>
</tr>
<tr>
<td>mintypecode (typechars[, type-set, default])</td>
<td>Return the character for the minimum-size type to which given types can be safely cast.</td>
</tr>
</tbody>
</table>

typename (char)

Return a description for the given data type code.

Parameters

char : str
Data type code.

**Returns**

* out : str

Description of the input data type code.

**See Also:**

dtype, typecodes

**Examples**

```python
>>> typechars = ['S1', '?', 'B', 'D', 'G', 'F', 'I', 'H', 'L', 'O', 'Q',
               's', 'U', 'V', 'b', 'd', 'g', 'f', 'i', 'h', 'l', 'q']
>>> for typechar in typechars:
...     print(typechar, ' : ', np.typename(typechar))
S1 : character
? : bool
B : unsigned char
D : complex double precision
G : complex long double precision
F : complex single precision
I : unsigned integer
H : unsigned short
L : unsigned long integer
O : object
Q : unsigned long long integer
S : string
U : unicode
V : void
b : signed char
d : double precision
g : long precision
f : single precision
i : integer
h : short
l : long integer
q : long long integer
```

**sctype2char** (*sc_type*)

Return the string representation of a scalar dtype.

**Parameters**

* sc_type : scalar dtype or object

If a scalar dtype, the corresponding string character is returned. If an object, *sctype2char* tries to infer its scalar type and then return the corresponding string character.

**Returns**

* typechar : str

The string character corresponding to the scalar type.

**Raises**

* ValueError :

  If *sc_type* is an object for which the type can not be inferred.

**See Also:**

3.4. Data type routines
Examples

```python
>>> for sctype in [np.int32, np.float, np.complex, np.string_, np.ndarray]:
...     print np.sctype2char(sctype)
1
d
D
S
O

>>> x = np.array([1., 2-1.j])
>>> np.sctype2char(x)
'D'
>>> np.sctype2char(list)
'O'
```

mintypecode (typechars, typeset='GDFgdf', default='d')
Return the character for the minimum-size type to which given types can be safely cast.

The returned type character must represent the smallest size dtype such that an array of the returned type can handle the data from an array of all types in typechars (or if typechars is an array, then its dtype.char).

Parameters

- **typechars**: list of str or array_like
  If a list of strings, each string should represent a dtype. If array_like, the character representation of the array dtype is used.

- **typeset**: str or list of str, optional
  The set of characters that the returned character is chosen from. The default set is 'GDFgdf'.

- **default**: str, optional
  The default character, this is returned if none of the characters in typechars matches a character in typeset.

Returns

- **typechar**: str
  The character representing the minimum-size type that was found.

See Also:
dtype, sctype2char, maximum_sctype

Examples

```python
>>> np.mintypecode(['d', 'f', 'S'])
'd'
>>> x = np.array([1.1, 2-3.j])
>>> np.mintypecode(x)
'D'
>>> np.mintypecode('abceh', default='G')
'G'
```
3.5 Input and output

3.5.1 NPZ files

<table>
<thead>
<tr>
<th>load(file[, mmap_mode])</th>
<th>Load a pickled, .npy, or .npz binary file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>save(file, arr)</td>
<td>Save an array to a binary file in NumPy .npy format.</td>
</tr>
<tr>
<td>savez(file, *args, **kwds)</td>
<td>Save several arrays into a single, compressed file in .npz format.</td>
</tr>
</tbody>
</table>

load(file, mmap_mode=None)

Load a pickled, .npy, or .npz binary file.

Parameters

**file** : file-like object or string

The file to read. It must support `seek()` and `read()` methods. If the filename extension is `.gz`, the file is first decompressed.

**mmap_mode** : {None, ‘r+’, ‘r’, ‘w+’, ‘c’}, optional

If not None, then memory-map the file, using the given mode (see `numpy.memmap`). The mode has no effect for pickled or zipped files. A memory-mapped array is stored on disk, and not directly loaded into memory. However, it can be accessed and sliced like any ndarray. Memory mapping is especially useful for accessing small fragments of large files without reading the entire file into memory.

Returns

**result** : array, tuple, dict, etc.

Data stored in the file.

Raises

**IOError** :

If the input file does not exist or cannot be read.

See Also:

`save`, `savez`, `loadtxt`

`memmap`

Create a memory-map to an array stored in a file on disk.

Notes

- If the file contains pickle data, then whatever is stored in the pickle is returned.
- If the file is a .npy file, then an array is returned.
- If the file is a .npz file, then a dictionary-like object is returned, containing `{filename: array}` key-value pairs, one for each file in the archive.

Examples

Store data to disk, and load it again:

```python
>>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
>>> np.load('/tmp/123.npy')
array([[1, 2, 3],
       [4, 5, 6]])
```

Mem-map the stored array, and then access the second row directly from disk:

```python
```
>>> X = np.load('/tmp/123.npy', mmap_mode='r')
>>> X[1, :]
memmap([4, 5, 6])

save (file, arr)
Save an array to a binary file in NumPy .npy format.

Parameters

  file : file or string
         File or filename to which the data is saved. If the filename does not already have a .npy extension, it is added.

  arr : array_like
         Array data to be saved.

See Also:

  savez
  Save several arrays into a .npz compressed archive

savetxt, load

Notes
For a description of the .npy format, see format.

Examples

>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()

>>> x = np.arange(10)
>>> np.save(outfile, x)

>>> outfile.seek(0) # only necessary in this example (with tempfile)
>>> np.load(outfile)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

savez (file, *args, **kwds)
Save several arrays into a single, compressed file in .npz format.

If keyword arguments are given, the names for variables assigned to the keywords are the keyword names (not the variable names in the caller). If arguments are passed in with no keywords, the corresponding variable names are arr_0, arr_1, etc.

Parameters

  file : str or file
         Either the file name (string) or an open file (file-like object) If file is a string, it names the output file. ".npz" will be appended to the file name if it is not already there.

  args : Arguments
         Any function arguments other than the file name are variables to save. Since it is not possible for Python to know their names outside savez, they will be saved with names "arr_0", "arr_1", and so on. These arguments can be any expression.
kwds : Keyword arguments

All keyword=value pairs cause the value to be saved with the name of the keyword.

See Also:

save
 Save a single array to a binary file in NumPy format.

savetxt
 Save an array to a file as plain text.

Notes

The .npz file format is a zipped archive of files named after the variables they contain. Each file contains one variable in .npy format. For a description of the .npy format, see format.

Examples

```python
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> x = np.arange(10)
>>> y = np.sin(x)

Using savez with *args, the arrays are saved with default names.

```python
>>> np.savez(outfile, x, y)
>>> outfile.seek(0)  # only necessary in this example (with tempfile)
>>> npz = np.load(outfile)
>>> npz.files
['arr_1', 'arr_0']
>>> npz['arr_0']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Using savez with **kwds, the arrays are saved with the keyword names.

```python
>>> outfile = TemporaryFile()
>>> np.savez(outfile, x=x, y=y)
>>> outfile.seek(0)
>>> npz = np.load(outfile)
>>> npz.files
['y', 'x']
>>> npz['x']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
3.5.2 Text files

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadtxt</td>
<td>Load data from a text file.</td>
</tr>
<tr>
<td>savetxt</td>
<td>Save an array to a text file.</td>
</tr>
<tr>
<td>genfromtxt</td>
<td>Load data from a text file, with missing values handled as specified.</td>
</tr>
<tr>
<td>fromregex</td>
<td>Construct an array from a text file, using regular expression parsing.</td>
</tr>
<tr>
<td>fromstring</td>
<td>Return a new 1-D array initialized from raw binary or text data in string.</td>
</tr>
<tr>
<td>ndarray.tofile</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>ndarray.tolist</td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
</tbody>
</table>

**loadtxt** (fname[, dtype, comments, ...])

Load data from a text file.

Each row in the text file must have the same number of values.

**Parameters**

- **fname**: file or str
  - File or filename to read. If the filename extension is .gz or .bz2, the file is first decompressed.

- **dtype**: dtype, optional
  - Data type of the resulting array. If this is a record data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.

- **comments**: str, optional
  - The character used to indicate the start of a comment.

- **delimiter**: str, optional
  - The string used to separate values. By default, this is any whitespace.

- **converters**: dict, optional
  - A dictionary mapping column number to a function that will convert that column to a float. E.g., if column 0 is a date string: converters = {0: datestr2num}. Converters can also be used to provide a default value for missing data: converters = {3: lambda s: float(s or 0)}.

- **skiprows**: int, optional
  - Skip the first skiprows lines.

- **usecols**: sequence, optional
  - Which columns to read, with 0 being the first. For example, usecols = (1, 4, 5) will extract the 2nd, 5th and 6th columns.

- **unpack**: bool, optional
  - If True, the returned array is transposed, so that arguments may be unpacked using x, y, z = loadtxt(...). Default is False.

**Returns**

- **out**: ndarray
  - Data read from the text file.
See Also:
load, fromstring, fromregex

scipy.io.loadmat
reads Matlab(R) data files

Examples

```python
>>> from StringIO import StringIO  # StringIO behaves like a file object
>>> c = StringIO("0 1\n2 3")
>>> np.loadtxt(c)
array([[ 0., 1.],
       [ 2., 3.]]))

>>> d = StringIO("M 21 72\nF 35 58")
>>> np.loadtxt(d, dtype=('gender', 'age', 'weight'),
             ... 'formats': ('S1', 'i4', 'f4'))
array([('M', 21, 72.0), ('F', 35, 58.0)],
      dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])

>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([ 1., 3.])
>>> y
array([ 2., 4.])
```

savetxt (fname, X, fmt='%.18e', delimiter=' ')
Save an array to a text file.

Parameters
fname : filename or file handle
If the filename ends in .gz, the file is automatically saved in compressed gzip format.
loadtxt understands gzipped files transparently.

X : array_like
Data to be saved to a text file.

fmt : str or sequence of strs
A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. ‘Iteration %d – %10.5f’, in which case delimiter is ignored.

delimiter : str
Character separating columns.

See Also:

save
Save an array to a binary file in NumPy .npy format

savez
Save several arrays into a .npz compressed archive
Notes

Further explanation of the fmt parameter (\%[flag]width[.precision]specifier):

flags:
- : left justify
+ : Forces to preceed result with + or -. 
0 : Left pad the number with zeros instead of space (see width).

width:
Minimum number of characters to be printed. The value is not truncated if it has more characters.

precision:

• For integer specifiers (eg. d, i, o, x), the minimum number of digits.
• For e, E and f specifiers, the number of digits to print after the decimal point.
• For g and G, the maximum number of significant digits.
• For s, the maximum number of characters.

specifiers:
- c: character
- d or i: signed decimal integer
- e or E: scientific notation with e or E.
- f: decimal floating point
- g, G: use the shorter of e, E or f
- o: signed octal
- s: string of characters
- u: unsigned decimal integer
- x, X: unsigned hexadecimal integer

This explanation of fmt is not complete, for an exhaustive specification see [R136].

References

[R136]

Examples

>>> savetxt('test.out', x, delimiter=',')    # X is an array
>>> savetxt('test.out', (x,y,z))             # x,y,z equal sized 1D arrays
>>> savetxt('test.out', x, fmt='%1.4e')     # use exponential notation

genfromtxt (fname, dtype=<type 'float'>, comments='#', delimiter=None, skiprows=0, skip_header=0, skip_footer=0, converters=None, missing='', missing_values=None, filling_values=None, usecols=None, names=None, excludelist=None, deletechars=None, autostrip=False, case_sensitive=True, defaultfmt='f%i', unpack=None, usemask=False, loose=True, invalid_raise=True)

Load data from a text file, with missing values handled as specified.

Each line past the first skiprows lines is split at the delimiter character, and characters following the comments character are discarded.
Parameters

fname : file or str
File or filename to read. If the filename extension is gz or bz2, the file is first decompressed.

dtype : dtype, optional
Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.

comments : str, optional
The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded.

delimiter : str, int, or sequence, optional
The string used to separate values. By default, any consecutive whitespaces act as delimiter. An integer or sequence of integers can also be provided as width(s) of each field.

skip_header : int, optional
The numbers of lines to skip at the beginning of the file.

skip_footer : int, optional
The numbers of lines to skip at the end of the file.

converters : variable or None, optional
The set of functions that convert the data of a column to a value. The converters can also be used to provide a default value for missing data: converters = {3: lambda s: float(s or 0)}.

missing_values : variable or None, optional
The set of strings corresponding to missing data.

filling_values : variable or None, optional
The set of values to be used as default when the data are missing.

usecols : sequence or None, optional
Which columns to read, with 0 being the first. For example, usecols = (1, 4, 5) will extract the 2nd, 5th and 6th columns.

names : {None, True, str, sequence}, optional
If names is True, the field names are read from the first valid line after the first skiprows lines. If names is a sequence or a single-string of comma-separated names, the names will be used to define the field names in a structured dtype. If names is None, the names of the dtype fields will be used, if any.

excludelist : sequence, optional
A list of names to exclude. This list is appended to the default list ['return','file','print']. Excluded names are appended an underscore: for example, file would become file_.

deletechars : str, optional
A string combining invalid characters that must be deleted from the names.

defaultfmt : str, optional
A format used to define default field names, such as “f%i” or “f_%02i”.

**autostrip** : bool, optional
Whether to automatically strip white spaces from the variables.

**case_sensitive** : {True, False, ‘upper’, ‘lower’}, optional
If True, field names are case sensitive. If False or ‘upper’, field names are converted to upper case. If ‘lower’, field names are converted to lower case.

**unpack** : bool, optional
If True, the returned array is transposed, so that arguments may be unpacked using x, y, z = loadtxt(...)

**usemask** : bool, optional
If True, return a masked array. If False, return a regular array.

**invalid_raise** : bool, optional
If True, an exception is raised if an inconsistency is detected in the number of columns. If False, a warning is emitted and the offending lines are skipped.

**Returns**
- **out** : ndarray
  Data read from the text file. If usemask is True, this is a masked array.

**See Also:**
- **numpy.loadtxt**
  equivalent function when no data is missing.

**Notes**
- When spaces are used as delimiters, or when no delimiter has been given as input, there should not be any missing data between two fields.
- When the variables are named (either by a flexible dtype or with names, there must not be any header in the file (else a ValueError exception is raised).
- Individual values are not stripped of spaces by default. When using a custom converter, make sure the function does remove spaces.

**Examples**

```python
>>> from StringIO import StringIO
>>> import numpy as np

Comma delimited file with mixed dtype

```
```python
>>> s.seek(0) # needed for StringIO example only
>>> data = np.genfromtxt(s, dtype=None,
    names = ['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, 'abcde'),
    dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', '|S5')])
```

Specifying dtype and names

```python
>>> s.seek(0)
>>> data = np.genfromtxt(s, dtype="i8,f8,S5",
    names=['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, 'abcde'),
    dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', '|S5')])
```

An example with fixed-width columns

```python
>>> s = StringIO("11.3abcde")
>>> data = np.genfromtxt(s, dtype=None, names=['intvar','fltvar','strvar'],
    delimiter=[1,3,5])
>>> data
array((1, 1.3, 'abcde'),
    dtype=[('intvar', '<i8'), ('fltvar', '<f8'), ('strvar', '|S5')])
```

fromregex (file, regexp, dtype)

Construct an array from a text file, using regular expression parsing.

The returned array is always a structured array, and is constructed from all matches of the regular expression in the file. Groups in the regular expression are converted to fields of the structured array.

Parameters

- **file** : str or file
  File name or file object to read.

- **regexp** : str or regexp
  Regular expression used to parse the file. Groups in the regular expression correspond to fields in the dtype.

- **dtype** : dtype or list of dtypes
  Dtype for the structured array.

Returns

- **output** : ndarray
  The output array, containing the part of the content of file that was matched by regexp. output is always a structured array.

Raises

- **TypeError** :
  When dtype is not a valid dtype for a structured array.

See Also:

fromstring, loadtxt

3.5. Input and output 397
Notes

Dtypes for structured arrays can be specified in several forms, but all forms specify at least the data type and field name. For details see `doc.structured_arrays`.

Examples

```python
>>> f = open('test.dat', 'w')
>>> f.write("1312 foo\n1534 bar\n444 qux")
>>> f.close()

>>> regexp = r"\d+\s+\S+"  # match [digits, whitespace, anything]
>>> output = np.fromregex('test.dat', regexp,
                          [('num', np.int64), ('key', '|S3')])
>>> output
array([(1312L, 'foo'), (1534L, 'bar'), (444L, 'qux')],
      dtype=[('num', '<i8'), ('key', '|S3')])
>>> output['num']
array([1312, 1534, 444], dtype=int64)
```

`fromstring()`

Return a new 1-D array initialized from raw binary or text data in string.

**Parameters**

- **string**: str
  A string containing the data.

- **dtype**: dtype, optional
  The data type of the array. For binary input data, the data must be in exactly this format.

- **count**: int, optional
  Read this number of `dtype` elements from the data. If this is negative, then the size will be determined from the length of the data.

- **sep**: str, optional
  If provided and not empty, then the data will be interpreted as ASCII text with decimal numbers. This argument is interpreted as the string separating numbers in the data. Extra whitespace between elements is also ignored.

**Returns**

- **arr**: array
  The constructed array.

**Raises**

ValueError:

If the string is not the correct size to satisfy the requested `dtype` and `count`.

**Examples**

```python
>>> np.fromstring('\x01\x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
```
```python
>>> np.fromstring('\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)

Invalid inputs:

```python
>>> np.fromstring('\x01\x02\x03\x04\x05', dtype=np.int32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: string size must be a multiple of element size
```n
```python
>>> np.fromstring('\x01\x02', dtype=np.uint8, count=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: string is smaller than requested size
```

tofile()
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.tostring())`.

- **format** : str
  Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes
This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

tolist()
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())`. 3.5. Input and output
3.5.3 String formatting

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array_repr</code></td>
<td>Return the string representation of an array.</td>
</tr>
<tr>
<td><code>array_str</code></td>
<td>Return a string representation of the data in an array.</td>
</tr>
</tbody>
</table>

`array_repr` (arr[, max_line_width, ...])
Return the string representation of an array.

Parameters

- **arr**: ndarray
  Input array.

- **max_line_width**: int, optional
  The maximum number of columns the string should span. Newline characters split the string appropriately after array elements.

- **precision**: int, optional
  Floating point precision. Default is the current printing precision (usually 8), which can be altered using `set_printoptions`.

- **suppress_small**: bool, optional
  Represent very small numbers as zero, default is False. Very small is defined by `precision`, if the precision is 8 then numbers smaller than 5e-9 are represented as zero.

Returns

- **string**: str
  The string representation of an array.

See Also:

- `array_str`, `array2string`, `set_printoptions`

Examples

```python
>>> np.array_repr(np.array([1,2]))
'array([1, 2])'
>>> np.array_repr(np.ma.array([0.]))
'MaskedArray([ 0.])'
>>> np.array_repr(np.array([], np.int32))
'array([], dtype=int32)'

>>> x = np.array([1e-6, 4e-7, 2, 3])
>>> np.array_repr(x, precision=6, suppress_small=True)
'array([ 0.000001, 0. , 2. , 3. ])'```
array_str(a, max_line_width=None, precision=None, suppress_small=None)

Return a string representation of the data in an array.

The data in the array is returned as a single string. This function is similar to array_repr, the difference is that array_repr also returns information on the type of array and data type.

Parameters
    a : ndarray
        Input array.
    max_line_width : int, optional
        Insert newlines if text is longer than max_line_width.
    precision : int, optional
        Floating point precision. Default is the current printing precision (usually 8), which can be altered using set_printoptions.
    suppress_small : bool, optional
        Represent very small numbers as zero, default is False. Very small is defined by precision, if the precision is 8 then numbers smaller than 5e-9 are represented as zero.

See Also:
array2string, array_repr, set_printoptions

Examples
>>> np.array_str(np.arange(3))
'[[0 1 2]]'

3.5.4 Memory mapping files

Create a memory-map to an array stored in a binary file on disk.

class memmap()

Create a memory-map to an array stored in a binary file on disk.

Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory. Numpy’s memmap’s are array-like objects. This differs from Python’s mmap module, which uses file-like objects.

Parameters
    filename : str or file-like object
        The file name or file object to be used as the array data buffer.
    dtype : data-type, optional
        The data-type used to interpret the file contents. Default is uint8.
    mode : {'r+', 'r', 'w+', 'c'}, optional
        The file is opened in this mode:
NumPy Reference, Release 1.4.1

| 'r' | Open existing file for reading only. |
| 'r+' | Open existing file for reading and writing. |
| 'w+' | Create or overwrite existing file for reading and writing. |
| 'c' | Copy-on-write: assignments affect data in memory, but changes are not saved to disk. The file on disk is read-only. |

Default is 'r+'.

**offset**: int, optional

In the file, array data starts at this offset. Since *offset* is measured in bytes, it should be a multiple of the byte-size of *dtype*. Requires shape=None. The default is 0.

**shape**: tuple, optional

The desired shape of the array. By default, the returned array will be 1-D with the number of elements determined by file size and data-type.

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

Specify the order of the ndarray memory layout: C (row-major) or Fortran (column-major). This only has an effect if the shape is greater than 1-D. The default order is 'C'.

**Notes**

The memmap object can be used anywhere an ndarray is accepted. Given a memmap `fp`, `isinstance(fp, numpy.ndarray)` returns `True`.

Memory-mapped arrays use the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

**Examples**

```python
>>> data = np.arange(12, dtype='float32')
>>> data.resize((3,4))
```

This example uses a temporary file so that doctest doesn’t write files to your directory. You would use a ‘normal’ filename.

```python
>>> from tempfile import mkdtemp
>>> import os.path as path

>>> filename = path.join(mkdtemp(), 'newfile.dat')
```

Create a memmap with dtype and shape that matches our data:

```python
>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
```

Write data to memmap array:

```python
>>> fp[:] = data[:]
```

```python
memmap([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]], dtype=float32)
```
Deletion flushes memory changes to disk before removing the object:

```python
>>> del fp
```

Load the memmap and verify data was stored:

```python
>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> newfp
memmap([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]], dtype=float32)
```

Read-only memmap:

```python
>>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> fpr.flags.writeable
False
```

Copy-on-write memmap:

```python
>>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3,4))
>>> fpc.flags.writeable
True
```

It’s possible to assign to copy-on-write array, but values are only written into the memory copy of the array, and not written to disk:

```python
>>> fpc
memmap([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]], dtype=float32)
>>> fpc[0,:] = 0
>>> fpc
memmap([[ 0.,  0.,  0.,  0.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]], dtype=float32)
```

File on disk is unchanged:

```python
>>> fpr
memmap([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]], dtype=float32)
```

Offset into a memmap:

```python
>>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
>>> fpo
memmap([ 4.,  5.,  6.,  7.,  8.,  9., 10., 11.], dtype=float32)
```

Methods
3.5.5 Text formatting options

```
set_printoptions ([precision, threshold, ...])  Set printing options.
get_printoptions ()  Return the current print options.
set_string_function (f[, repr])  Set a Python function to be used when pretty printing arrays.
```

```
set_printoptions (precision=None, threshold=None, edgeitems=None, linewidth=None, suppress=None, nanstr=None, infstr=None)
```

Set printing options.

These options determine the way floating point numbers, arrays and other NumPy objects are displayed.

**Parameters**

- `precision` : int, optional
  Number of digits of precision for floating point output (default 8).

- `threshold` : int, optional
  Total number of array elements which trigger summarization rather than full repr (default 1000).

- `edgeitems` : int, optional
  Number of array items in summary at beginning and end of each dimension (default 3).

- `linewidth` : int, optional
  The number of characters per line for the purpose of inserting line breaks (default 75).

- `suppress` : bool, optional
  Whether or not suppress printing of small floating point values using scientific notation (default False).

- `nanstr` : str, optional
  String representation of floating point not-a-number (default NaN).

- `infstr` : str, optional
  String representation of floating point infinity (default Inf).

**See Also:**

- `get_printoptions`
- `set_string_function`

**Examples**

Floating point precision can be set:

```
>>> np.set_printoptions(precision=4)
>>> print np.array([1.123456789])
[ 1.1235]
```

Long arrays can be summarised:

```
>>> np.set_printoptions(threshold=5)
>>> print np.arange(10)
[ 0 1 2 ..., 7 8 9]
```

Small results can be suppressed:
>>> eps = np.finfo(float).eps
>>> x = np.arange(4.)
>>> x**2 - (x + eps)**2
array([-4.9304e-32, -4.4409e-16, 0.0000e+00, 0.0000e+00])

>>> np.set_printoptions(suppress=True)
>>> x**2 - (x + eps)**2
array([-0., -0., 0., 0.])

get_printoptions()
Return the current print options.

Returns
print_opts : dict
   Dictionary of current print options with keys
   • precision : int
   • threshold : int
   • edgeitems : int
   • linewidth : int
   • suppress : bool
   • nanstr : str
   • infstr : str
   For a full description of these options, see set_printoptions.

See Also:
set_printoptions, set_string_function

set_string_function()
Set a Python function to be used when pretty printing arrays.

Parameters
f : function or None
   Function to be used to pretty print arrays. The function should expect a single array argument and return a string of the representation of the array. If None, the function is reset to the default NumPy function to print arrays.

repr : bool, optional
   If True (default), the function for pretty printing (__repr__) is set, if False the function that returns the default string representation (__str__) is set.

See Also:
set_printoptions, get_printoptions

Examples

>>> def pprint(arr):
...     return 'HA! - What are you going to do now?'
...
>>> np.set_string_function(pprint)
>>> a = np.arange(10)
>>> a

3.5. Input and output
HA! - What are you going to do now?

```python
>>> print a
[0 1 2 3 4 5 6 7 8 9]
```

We can reset the function to the default:

```python
>>> np.set_string_function(None)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 'l')
```

`repr` affects either pretty printing or normal string representation. Note that `__repr__` is still affected by setting `__str__` because the width of each array element in the returned string becomes equal to the length of the result of `__str__()`.

```python
>>> x = np.arange(4)
>>> np.set_string_function(lambda x: 'random', repr=False)
>>> x.__str__()  # 'random'
>>> x.__repr__()  # 'array([0, 1, 2, 3])'
```

### 3.5.6 Base-n representations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binary_repr</code></td>
<td>Return the binary representation of the input number as a string.</td>
</tr>
<tr>
<td><code>base_repr</code></td>
<td>Return a string representation of a number in the given base system.</td>
</tr>
</tbody>
</table>

**`binary_repr`**

Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two’s complement of the number is returned, with respect to that width.

In a two’s-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers [R23]. A N-bit two’s-complement system can represent every integer in the range $-2^{N-1}$ to $+2^{N-1} - 1$.

**Parameters**

- `num` : int
  - Only an integer decimal number can be used.

- `width` : int, optional
  - The length of the returned string if `num` is positive, the length of the two’s complement if `num` is negative.

**Returns**

- `bin` : str
  - Binary representation of `num` or two’s complement of `num`.

**See Also:**

**`base_repr`**

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

`binary_repr` is equivalent to using `base_repr` with base 2, but about 25x faster.

References

[R23]

Examples

```python
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

The two’s complement is returned when the input number is negative and width is specified:

```python
>>> np.binary_repr(-3, width=4)
'1101'
```

`base_repr(number, base=2, padding=0)`

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

Parameters

- `number`: scalar
  - The value to convert. Only positive values are handled.
- `base`: int, optional
  - Convert `number` to the `base` number system. The valid range is 2-36, the default value is 2.
- `padding`: int, optional
  - Number of zeros padded on the left. Default is 0 (no padding).

Returns

- `out`: str
  - String representation of `number` in `base` system.

See Also:

`binary_repr`

Faster version of `base_repr` for base 2 that also handles negative numbers.

Examples

```python
>>> np.base_repr(5)
'101'
>>> np.base_repr(6, 5)
'11'
>>> np.base_repr(7, base=5, padding=3)
'00012'
```
```python
>>> np.base_repr(10, base=16)
'A'
>>> np.base_repr(32, base=16)
'20'
```

### 3.5.7 Data sources

**DataSource** ([destpath])  A generic data source file (file, http, ftp,...).

class **DataSource** (destpath=')
A generic data source file (file, http, ftp,...).

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

**Parameters**

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

**Notes**

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

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

Temporary directories are deleted when the DataSource is deleted.

**Examples**

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

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

**Methods**
3.6 Discrete Fourier Transform (numpy.fft)

3.6.1 Standard FFTs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fft</code></td>
<td>Compute the one-dimensional discrete Fourier Transform.</td>
</tr>
<tr>
<td><code>ifft</code></td>
<td>Compute the one-dimensional inverse discrete Fourier Transform.</td>
</tr>
<tr>
<td><code>fft2</code></td>
<td>Compute the 2-dimensional discrete Fourier Transform.</td>
</tr>
<tr>
<td><code>ifft2</code></td>
<td>Compute the 2-dimensional inverse discrete Fourier Transform.</td>
</tr>
<tr>
<td><code>fftn</code></td>
<td>Compute the N-dimensional discrete Fourier Transform.</td>
</tr>
<tr>
<td><code>ifftn</code></td>
<td>Compute the N-dimensional inverse discrete Fourier Transform.</td>
</tr>
</tbody>
</table>

**fft**($a$, $n=$None, $axis$=-1)

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional \( n \)-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

**Parameters**

- **a**: array_like
  - Input array, can be complex.
- **n**: int, optional
  - Length of the transformed axis of the output. If \( n \) is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If \( n \) is not given, the length of the input (along the axis specified by \( axis \)) is used.
- **axis**: int, optional
  - Axis over which to compute the FFT. If not given, the last axis is used.

**Returns**

- **out**: complex ndarray
  - The truncated or zero-padded input, transformed along the axis indicated by \( axis \), or the last one if \( axis \) is not specified.

**Raises**

- **IndexError** :
  - If \( axes \) is larger than the last axis of \( a \).

**See Also**:

- **numpy.fft**
  - for definition of the DFT and conventions used.
- **ifft**
  - The inverse of \( fft \).
- **fft2**
  - The two-dimensional FFT.
- **fftn**
  - The \( n \)-dimensional FFT.
- **rfftn**
  - The \( n \)-dimensional FFT of real input.
- **fftfreq**
  - Frequency bins for given FFT parameters.
Notes

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when \( n \) is a power of 2, and the transform is therefore most efficient for these sizes.

The DFT is defined, with the conventions used in this implementation, in the documentation for the `numpy.fft` module.

References

[CT]

Examples

```python
>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
array([ -3.44505240e-16 +1.14383329e-17j,
   8.00000000e+00 -5.71092652e-15j,
   2.33482938e-16 +1.22460635e-16j,
   1.64863782e-15 +1.77635684e-15j,
   9.95839695e-17 +2.33482938e-16j,
   0.00000000e+00 +1.66837030e-15j,
   1.14383329e-17 +1.22460635e-16j,
  -1.64863782e-15 +1.77635684e-15j])
```

```python
>>> import matplotlib.pyplot as plt
>>> t = np.arange(256)
>>> sp = np.fft.fft(np.sin(t))
>>> freq = np.fft.fftfreq(t.shape[-1])
>>> plt.plot(freq, sp.real, freq, sp.imag)
>>> plt.show()
```

In this example, real input has an FFT which is Hermitian, i.e., symmetric in the real part and anti-symmetric in the imaginary part, as described in the `numpy.fft` documentation.

```
```

### ifft

```
```

```python
ifft(a, n=None, axis=-1)
```

Compute the one-dimensional inverse discrete Fourier Transform.
This function computes the inverse of the one-dimensional $n$-point discrete Fourier transform computed by `fft`. In other words, \( \text{ifft}(\text{fft}(a)) == a \) to within numerical accuracy. For a general description of the algorithm and definitions, see `numpy.fft`.

The input should be ordered in the same way as is returned by `fft`, i.e., \( a[0] \) should contain the zero-frequency term, \( a[1:\lfloor n/2+1 \rfloor] \) should contain the positive-frequency terms, and \( a[\lfloor n/2+1 \rfloor:] \) should contain the negative-frequency terms, in order of decreasingly negative frequency. See `numpy.fft` for details.

**Parameters**

- **a**: array_like
  - Input array, can be complex.

- **n**: int, optional
  - Length of the transformed axis of the output. If \( n \) is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If \( n \) is not given, the length of the input (along the axis specified by `axis`) is used. See notes about padding issues.

- **axis**: int, optional
  - Axis over which to compute the inverse DFT. If not given, the last axis is used.

**Returns**

- **out**: complex ndarray
  - The truncated or zero-padded input, transformed along the axis indicated by `axis`, or the last one if `axis` is not specified.

**Raises**

- **IndexError**
  - If `axes` is larger than the last axis of `a`.

**See Also**

- `numpy.fft`
  - An introduction, with definitions and general explanations.

- `fft`
  - The one-dimensional (forward) FFT, of which `ifft` is the inverse.

- `ifft2`
  - The two-dimensional inverse FFT.

- `ifftn`
  - The n-dimensional inverse FFT.

**Notes**

If the input parameter \( n \) is larger than the size of the input, the input is padded by appending zeros at the end. Even though this is the common approach, it might lead to surprising results. If a different padding is desired, it must be performed before calling `ifft`.

**Examples**

```python
>>> np.fft.ifft([0, 4, 0, 0])
array([ 1.+0.j, 0.+1.j, -1.+0.j, 0.-1.j])
```

Create and plot a band-limited signal with random phases:
```python
>>> import matplotlib.pyplot as plt
>>> t = np.arange(400)
>>> n = np.zeros((400,), dtype=complex)
>>> n[40:60] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20,)))
>>> s = np.fft.ifft(n)
>>> plt.plot(t, s.real, 'b-', t, s.imag, 'r--')
>>> plt.legend(('real', 'imaginary'))
>>> plt.show()
```

fft2(a, s=None, axes=(-2, -1))

Compute the 2-dimensional discrete Fourier Transform

This function computes the \(n\)-dimensional discrete Fourier Transform over any axes in an \(M\)-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

Parameters

a : array_like
    Input array, can be complex

s : sequence of ints, optional
    Shape (length of each transformed axis) of the output \((s[0])\) refers to axis 0, \((s[1])\) to axis 1, etc.). This corresponds to \(n\) for \(fft(x, n)\). Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if \(s\) is not given, the shape of the input (along the axes specified by \(axes\)) is used.

axes : sequence of ints, optional
    Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in \(axes\) means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

Returns

out : complex ndarray
    The truncated or zero-padded input, transformed along the axes indicated by \(axes\), or the last two axes if \(axes\) is not given.
Raises

* ValueError:
  
  If `s` and `axes` have different length, or `axes` not given and `len(s) != 2`.

* IndexError:
  
  If an element of `axes` is larger than than the number of axes of `a`.

See Also:

* `numpy.fft`
  
  Overall view of discrete Fourier transforms, with definitions and conventions used.

* `ifft2`
  
  The inverse two-dimensional FFT.

* `fft`
  
  The one-dimensional FFT.

* `fftn`
  
  The n-dimensional FFT.

* `fftsift`
  
  Shifts zero-frequency terms to the center of the array. For two-dimensional input, swaps first and third quadrants, and second and fourth quadrants.

Notes

`ifft2` is just `fftn` with a different default for `axes`.

The output, analogously to `fft`, contains the term for zero frequency in the low-order corner of the transformed axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of the axes, in order of decreasingly negative frequency.

See `fftn` for details and a plotting example, and `numpy.fft` for definitions and conventions used.

Examples

```python
>>> a = np.mgrid[:5, :5][0]
>>> np.fft.fft2(a)
array([[ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [ 5.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [10.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [15.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [20.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j]])
```

```python
ifft2(a, s=None, axes=(-2, -1))
```

Compute the 2-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the 2-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifft2(fftt2(a)) == a` to within numerical accuracy. By default, the inverse transform is computed over the last two axes of the input array.

The input, analogously to `fft`, should be ordered in the same way as is returned by `fftt2`, i.e. it should have the term for zero frequency in the low-order corner of the two axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of both axes, in order of decreasingly negative frequency.
### Parameters

**a** : array_like

Input array, can be complex.

**s** : sequence of ints, optional

Shape (length of each axis) of the output (s[0] refers to axis 0, s[1] to axis 1, etc.). This corresponds to n for ifft(x, n). Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If s is not given, the shape of the input (along the axes specified by axes) is used. See notes for issue on ifft zero padding.

**axes** : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in axes means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

### Returns

**out** : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by axes, or the last two axes if axes is not given.

### Raises

**ValueError** :

If s and axes have different length, or axes not given and len(s) != 2.

**IndexError** :

If an element of axes is larger than than the number of axes of a.

### See Also:

- **numpy.fft**
  
  Overall view of discrete Fourier transforms, with definitions and conventions used.

- **fft2**
  
  The forward 2-dimensional FFT, of which ifft2 is the inverse.

- **ifftn**
  
  The inverse of the n-dimensional FFT.

- **fft**
  
  The one-dimensional FFT.

- **ifft**
  
  The one-dimensional inverse FFT.

### Notes

ifft2 is just ifftn with a different default for axes.

See ifftn for details and a plotting example, and numpy.fft for definition and conventions used.

Zero-padding, analogously with ifft, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before ifft2 is called.
**Examples**

```python
>>> a = 4 * np.eye(4)
>>> np.fft.ifft2(a)
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j]])
```

**fftn** *(a, s=None, axes=None)*

Compute the N-dimensional discrete Fourier Transform.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT).

**Parameters**

- **a**: array_like
  - Input array, can be complex.

- **s**: sequence of ints, optional
  - Shape (length of each transformed axis) of the output (s[0] refers to axis 0, s[1] to axis 1, etc.). This corresponds to n for fft(x, n). Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros.
  - If s is not given, the shape of the input (along the axes specified by axes) is used.

- **axes**: sequence of ints, optional
  - Axes over which to compute the FFT. If not given, the last len(s) axes are used, or all axes if s is also not specified. Repeated indices in axes means that the transform over that axis is performed multiple times.

**Returns**

- **out**: complex ndarray
  - The truncated or zero-padded input, transformed along the axes indicated by axes, or by a combination of s and a, as explained in the parameters section above.

**Raises**

- **ValueError**
  - If s and axes have different length.

- ** IndexError**
  - If an element of axes is larger than than the number of axes of a.

**See Also**:

- **numpy.fft**
  - Overall view of discrete Fourier transforms, with definitions and conventions used.

- **ifftn**
  - The inverse of fftn, the inverse n-dimensional FFT.

- **fft**
  - The one-dimensional FFT, with definitions and conventions used.

- **rfft**
  - The n-dimensional FFT of real input.
**fft2**

The two-dimensional FFT.

**fftshift**

Shifts zero-frequency terms to centre of array

**Notes**

The output, analogously to `fft`, contains the term for zero frequency in the low-order corner of all axes, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

See `numpy.fft` for details, definitions and conventions used.

**Examples**

`ifftn(a, s=None, axes=None)`

Compute the N-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifftn(fftn(a)) == a` to within numerical accuracy. For a description of the definitions and conventions used, see `numpy.fft`.

The input, analogously to `iff`, should be ordered in the same way as is returned by `fftn`, i.e. it should have the term for zero frequency in all axes in the low-order corner, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

**Parameters**

- `a` : array_like
  Input array, can be complex.

- `s` : sequence of ints, optional
  Shape (length of each transformed axis) of the output (s[0] refers to axis 0, s[1] to axis 1, etc.). This corresponds to n for `ifft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if s is not given, the shape of the input (along the axes specified by `axes`) is used. See notes for issue on ifft zero padding.

- `axes` : sequence of ints, optional
  Axes over which to compute the IFFT. If not given, the last `len(s)` axes are used, or all axes if `s` is also not specified. Repeated indices in `axes` means that the inverse transform over that axis is performed multiple times.

**Returns**

- `out` : complex ndarray
  The truncated or zero-padded input, transformed along the axes indicated by `axes`, or by a combination of `s` or `a`, as explained in the parameters section above.

**Raises**

- `ValueError` : If `s` and `axes` have different length.

- `IndexError` : If an element of `axes` is larger than than the number of axes of `a`.

**See Also:**
numpy.fft

Overall view of discrete Fourier transforms, with definitions and conventions used.

fftn

The forward \( n \)-dimensional FFT, of which ifftn is the inverse.

ifft

The one-dimensional inverse FFT.

ifft2

The two-dimensional inverse FFT.

ifftshift

Undoes fftshift, shifts zero-frequency terms to beginning of array.

Notes

See numpy.fft for definitions and conventions used.

Zero-padding, analogously with ifft, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before ifftn is called.

Examples

```python
>>> a = np.eye(4)
>>> np.fft.ifftn(np.fft.fftn(a, axes=(0,)), axes=(1,))
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
```

Create and plot an image with band-limited frequency content:

```python
>>> import matplotlib.pyplot as plt
>>> n = np.zeros((200,200), dtype=complex)
>>> n[60:80, 20:40] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20, 20)))
>>> im = np.fft.ifftn(n).real
>>> plt.imshow(im)
>>> plt.show()
```
### 3.6.2 Real FFTs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rfft(a, n[, axis])</code></td>
<td>Compute the one-dimensional discrete Fourier Transform for real input.</td>
</tr>
<tr>
<td><code>irfft(a[, n, axis])</code></td>
<td>Compute the inverse of the n-point DFT for real input.</td>
</tr>
<tr>
<td><code>rfft2(a[, s, axes, -1])</code></td>
<td>Compute the 2-dimensional FFT of a real array.</td>
</tr>
<tr>
<td><code>irfft2(a[, s, axes, -1])</code></td>
<td>Compute the 2-dimensional inverse FFT of a real array.</td>
</tr>
<tr>
<td><code>rfftn(a[, s, axes])</code></td>
<td>Compute the N-dimensional discrete Fourier Transform for real input.</td>
</tr>
<tr>
<td><code>irfftn(a[, s, axes])</code></td>
<td>Compute the inverse of the N-dimensional FFT of real input.</td>
</tr>
</tbody>
</table>

**`rfft(a, n=None, axis=-1)`**

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional $n$-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

**Parameters**

- `a` : array_like
  - Input array
- `n` : int, optional
  - Number of points along transformation axis in the input to use. If $n$ is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If $n$ is not given, the length of the input (along the axis specified by `axis`) is used.
- `axis` : int, optional
  - Axis over which to compute the FFT. If not given, the last axis is used.

**Returns**

- `out` : complex ndarray
  - The truncated or zero-padded input, transformed along the axis indicated by `axis`, or the last one if `axis` is not specified. The length of the transformed axis is $n/2+1$.

**Raises**

- `IndexError`
  - If `axis` is larger than the last axis of $a$. 

See Also:

numpy.fft
   For definition of the DFT and conventions used.
irfft
   The inverse of rfft.
fft
   The one-dimensional FFT of general (complex) input.
fftn
   The n-dimensional FFT.
rfftn
   The n-dimensional FFT of real input.

Notes
When the DFT is computed for purely real input, the output is Hermite-symmetric, i.e. the negative frequency
terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency
terms are therefore redundant. This function does not compute the negative frequency terms, and the length of
the transformed axis of the output is therefore n/2+1.
When A = rfft(a), A[0] contains the zero-frequency term, which must be purely real due to the Hermite
symmetry.
If n is even, A[-1] contains the term for frequencies n/2 and -n/2, and must also be purely real. If n is odd,
A[-1] contains the term for frequency A[(n-1)/2], and is complex in the general case.
If the input a contains an imaginary part, it is silently discarded.

Examples
>>> np.fft.fft([0, 1, 0, 0])
array([ 1.+0.j, 0.-1.j, -1.+0.j, 0.+1.j])
>>> np.fft.rfft([0, 1, 0, 0, 0])
array([ 1.+0.j, 0.-1.j, -1.+0.j])

Notice how the final element of the fft output is the complex conjugate of the second element, for real input. For
rfft, this symmetry is exploited to compute only the non-negative frequency terms.

irfft(a, n=None, axis=-1)
Compute the inverse of the n-point DFT for real input.
This function computes the inverse of the one-dimensional n-point discrete Fourier Transform of real input
computed by rfft. In other words, irfft(rfft(a), len(a)) == a to within numerical accuracy. (See
Notes below for why len(a) is necessary here.)
The input is expected to be in the form returned by rfft, i.e. the real zero-frequency term followed by the complex
positive frequency terms in order of increasing frequency. Since the discrete Fourier Transform of real input is
Hermite-symmetric, the negative frequency terms are taken to be the complex conjugates of the corresponding
positive frequency terms.

Parameters
   a : array_like
       The input array.
   n : int, optional
Length of the transformed axis of the output. For \( n \) output points, \( n/2+1 \) input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If \( n \) is not given, it is determined from the length of the input (along the axis specified by \( \text{axis} \)).

\( \text{axis} \) : int, optional

Axis over which to compute the inverse FFT.

Returns

\( \text{out} \) : ndarray

The truncated or zero-padded input, transformed along the axis indicated by \( \text{axis} \), or the last one if \( \text{axis} \) is not specified. The length of the transformed axis is \( n \), or, if \( n \) is not given, \( 2*(m-1) \) where \( m \) is the length of the transformed axis of the input. To get an odd number of output points, \( n \) must be specified.

Raises

\( \text{IndexError} \) :

If \( \text{axis} \) is larger than the last axis of \( a \).

See Also:

\texttt{numpy.fft}

For definition of the DFT and conventions used.

\texttt{rfft}

The one-dimensional FFT of real input, of which \texttt{irfft} is inverse.

\texttt{fft}

The one-dimensional FFT.

\texttt{irfft2}

The inverse of the two-dimensional FFT of real input.

\texttt{irfftn}

The inverse of the \( n \)-dimensional FFT of real input.

Notes

Returns the real valued \( n \)-point inverse discrete Fourier transform of \( a \), where \( a \) contains the non-negative frequency terms of a Hermite-symmetric sequence. \( n \) is the length of the result, not the input.

If you specify an \( n \) such that \( a \) must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to \( m \) points via Fourier interpolation by:

\[ a_{\text{resamp}} = \text{irfft}(\text{rfft}(a), m). \]

Examples

```python
>>> np.fft.ifft([1, -1j, -1, 1j])
array([ 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j])
>>> np.fft.irfft([1, -1j, -1])
array([ 0., 1., 0., 0.])
```

Notice how the last term in the input to the ordinary \texttt{ifft} is the complex conjugate of the second term, and the output has zero imaginary part everywhere. When calling \texttt{irfft}, the negative frequencies are not specified, and the output array is purely real.

\texttt{rfft2}(\( a, s=\text{None}, \text{axes}=(-2, -1) \))

Compute the 2-dimensional FFT of a real array.
Parameters
   a : array
       Input array, taken to be real.
   s : sequence of ints, optional
       Shape of the FFT.
   axes : sequence of ints, optional
       Axes over which to compute the FFT.

Returns
   out : ndarray
       The result of the real 2-D FFT.

See Also:

rfftn
Compute the N-dimensional discrete Fourier Transform for real input.

Notes
This is really just rfftn with different default behavior. For more details see rfftn.

irfft2(a, s=None, axes=(-2, -1))
Compute the 2-dimensional inverse FFT of a real array.

Parameters
   a : array_like
       The input array
   s : sequence of ints, optional
       Shape of the inverse FFT.
   axes : sequence of ints, optional
       The axes over which to compute the inverse fft. Default is the last two axes.

Returns
   out : ndarray
       The result of the inverse real 2-D FFT.

See Also:

irfftn
Compute the inverse of the N-dimensional FFT of real input.

Notes
This is really irfftn with different defaults. For more details see irfftn.

rfftn(a, s=None, axes=None)
Compute the N-dimensional discrete Fourier Transform for real input.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-
dimensional real array by means of the Fast Fourier Transform (FFT). By default, all axes are transformed,
with the real transform performed over the last axis, while the remaining transforms are complex.
**Parameters**

- **a**: array_like
  
  Input array, taken to be real.

- **s**: sequence of ints, optional
  
  Shape (length along each transformed axis) to use from the input. ($s[0]$ refers to axis 0, $s[1]$ to axis 1, etc.). The final element of $s$ corresponds to $n$ for $rfft(x, n)$, while for the remaining axes, it corresponds to $n$ for $fft(x, n)$. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If $s$ is not given, the shape of the input (along the axes specified by $axes$) is used.

- **axes**: sequence of ints, optional
  
  Axes over which to compute the FFT. If not given, the last $\text{len}(s)$ axes are used, or all axes if $s$ is also not specified.

**Returns**

- **out**: complex ndarray
  
  The truncated or zero-padded input, transformed along the axes indicated by $axes$, or by a combination of $s$ and $a$, as explained in the parameters section above. The length of the last axis transformed will be $s[-1]/2+1$, while the remaining transformed axes will have lengths according to $s$, or unchanged from the input.

**Raises**

- **ValueError**: If $s$ and $axes$ have different length.

- **IndexError**: If an element of $axes$ is larger than than the number of axes of $a$.

**See Also**

- **irfftn**: The inverse of $rfftn$, i.e. the inverse of the n-dimensional FFT of real input.

- **fft**: The one-dimensional FFT, with definitions and conventions used.

- **rfft**: The one-dimensional FFT of real input.

- **fftn**: The n-dimensional FFT.

- **rfft2**: The two-dimensional FFT of real input.

**Notes**

The transform for real input is performed over the last transformation axis, as by $rfft$, then the transform over the remaining axes is performed as by $fftn$. The order of the output is as for $rfft$ for the final transformation axis, and as for $fftn$ for the remaining transformation axes.

See $fft$ for details, definitions and conventions used.
Examples

```python
>>> a = np.ones((2, 2, 2))
>>> np.fft.rfftn(a)
array([[ 8.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j]])

>>> np.fft.rfftn(a, axes=(2, 0))
array([[ 4.+0.j,  0.+0.j],
       [ 4.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j]])
```

irfftn(a, s=None, axes=None)

Compute the inverse of the N-dimensional FFT of real input.

This function computes the inverse of the N-dimensional discrete Fourier Transform for real input over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, \texttt{irfftn(rfftn(a), a.shape)} == \texttt{a} to within numerical accuracy. (The \texttt{a.shape} is necessary like \texttt{len(a)} is for \texttt{irfft}, and for the same reason.)

The input should be ordered in the same way as is returned by \texttt{rfftn}, i.e. as for \texttt{irfft} for the final transformation axis, and as for \texttt{ifftn} along all the other axes.

Parameters

- \texttt{a} : array_like
  - Input array.
- \texttt{s} : sequence of ints, optional
  - Shape (length of each transformed axis) of the output (\texttt{s[0]} refers to axis 0, \texttt{s[1]} to axis 1, etc.). \texttt{s} is also the number of input points used along this axis, except for the last axis, where \texttt{s[-1]}//2+1 points of the input are used. Along any axis, if the shape indicated by \texttt{s} is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If \texttt{s} is not given, the shape of the input (along the axes specified by \texttt{axes}) is used.
- \texttt{axes} : sequence of ints, optional
  - Axes over which to compute the inverse FFT. If not given, the last \texttt{len(s)} axes are used, or all axes if \texttt{s} is also not specified. Repeated indices in \texttt{axes} means that the inverse transform over that axis is performed multiple times.

Returns

- \texttt{out} : ndarray
  - The truncated or zero-padded input, transformed along the axes indicated by \texttt{axes}, or by a combination of \texttt{s} or \texttt{a}, as explained in the parameters section above. The length of each transformed axis is as given by the corresponding element of \texttt{s}, or the length of the input in every axis except for the last one if \texttt{s} is not given. In the final transformed axis the length of the output when \texttt{s} is not given is \texttt{2*(m-1)} where \texttt{m} is the length of the final transformed axis of the input. To get an odd number of output points in the final axis, \texttt{s} must be specified.

Raises

- \texttt{ValueError}
If `s` and `axes` have different length.

**IndexError**:

If an element of `axes` is larger than than the number of axes of `a`.

**See Also:**

- `rfftn`
  - The forward n-dimensional FFT of real input, of which `irfftn` is the inverse.
- `fft`
  - The one-dimensional FFT, with definitions and conventions used.
- `irfft`
  - The inverse of the one-dimensional FFT of real input.
- `irfft2`
  - The inverse of the two-dimensional FFT of real input.

**Notes**

See `fft` for definitions and conventions used.

See `rfft` for definitions and conventions used for real input.

**Examples**

```python
>>> a = np.zeros((3, 2, 2))
>>> a[0, 0, 0] = 3 * 2 * 2
>>> np.fft.irfftn(a)
array([[[ 1.,  1.],
        [ 1.,  1.]],
       [[[ 1.,  1.],
        [ 1.,  1.]],
       [[[ 1.,  1.],
        [ 1.,  1.]]]])
```

### 3.6.3 Hermitian FFTs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hfft</code></td>
<td>Compute the FFT of a signal whose spectrum has Hermitian symmetry.</td>
</tr>
<tr>
<td><code>ihfft</code></td>
<td>Compute the inverse FFT of a signal whose spectrum has Hermitian symmetry.</td>
</tr>
</tbody>
</table>

**`hfft`**: `hfft(a, n=None, axis=-1)`

Compute the FFT of a signal whose spectrum has Hermitian symmetry.

**Parameters**

- `a`: array_like
  - The input array.
- `n`: int, optional
  - The length of the FFT.
- `axis`: int, optional
  - The axis over which to compute the FFT, assuming Hermitian symmetry of the spectrum. Default is the last axis.
Returns

\[ \text{out} : \text{ndarray} \]

The transformed input.

See Also:

- **rfft**
  
  Compute the one-dimensional FFT for real input.

- **ihfft**
  
  The inverse of hfft.

Notes

hfft/ihfft are a pair analogous to rfft/irfft, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it’s hfft for which you must supply the length of the result if it is to be odd: ihfft(hfft(a), len(a)) == a, within numerical accuracy.

Examples

```python
>>> signal = np.array([[1, 1.j], [-1.j, 2]])
>>> np.conj(signal.T) - signal  # check Hermitian symmetry
array([[ 0.-0.j, 0.+0.j],
       [ 0.+0.j, 0.-0.j]])
>>> freq_spectrum = np.fft.hfft(signal)
>>> freq_spectrum
array([[ 1., 1.],
       [ 2., -2.]])
```

ihfft \(a, n=None, axis=-1\)

Compute the inverse FFT of a signal whose spectrum has Hermite symmetry.

Parameters

- **a** : array_like
  
  Input array.

- **n** : int, optional
  
  Length of the inverse FFT.

- **axis** : int, optional
  
  Axis over which to compute the inverse FFT, assuming Hermitian symmetry of the spectrum. Default is the last axis.

Returns

- **out** : ndarray
  
  The transformed input.

See Also:

hfft, irfft

Notes

hfft/ihfft are a pair analogous to rfft/irfft, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it’s hfft for which you must supply the length of the result if it is to be odd: ihfft(hfft(a), len(a)) == a, within numerical accuracy.
### 3.6.4 Helper routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fftfreq(n, d=1.0)</code></td>
<td>Return the Discrete Fourier Transform sample frequencies.</td>
</tr>
<tr>
<td><code>fftshift(x[, axes])</code></td>
<td>Shift the zero-frequency component to the center of the spectrum.</td>
</tr>
<tr>
<td><code>ifftshift(x[, axes])</code></td>
<td>The inverse of <code>fftshift</code>.</td>
</tr>
</tbody>
</table>

#### `fftfreq(n, d=1.0)`

Return the Discrete Fourier Transform sample frequencies.

The returned float array contains the frequency bins in cycles/unit (with zero at the start) given a window length \( n \) and a sample spacing \( d \):

\[
\begin{align*}
  f &= \left[ 0, 1, \ldots, \frac{n-1}{2}, -\frac{n-1}{2}, \ldots, -1 \right] / (d \cdot n) & \text{if } n \text{ is even} \\
  f &= \left[ 0, 1, \ldots, \frac{n-1}{2}, -\frac{n-1}{2}, \ldots, -1 \right] / (d \cdot n) & \text{if } n \text{ is odd}
\end{align*}
\]

**Parameters**

- **\( n \)**: int
  - Window length.
- **\( d \)**: scalar
  - Sample spacing.

**Returns**

- **\( \text{out} \)**: ndarray
  - The array of length \( n \), containing the sample frequencies.

#### Examples

```python
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0. , 1.25, 2.5 , 3.75, -5. , -3.75, -2.5 , -1.25])
```

#### `fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that \( y[0] \) is the Nyquist component only if \( \text{len}(x) \) is even.

**Parameters**

- **\( x \)**: array_like
  - Input array.
- **\( \text{axes} \)**: int or shape tuple, optional
  - Axes over which to shift. Default is None, which shifts all axes.

**Returns**

- **\( y \)**: ndarray
  - The shifted array.

**See Also:**
**ifftshift**

The inverse of \textit{fftshift}.

**Examples**

```python
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0., 1., 2., 3., 4., -5., -4., -3., -2., -1.])
```

```python
>>> np.fft.ifftshift(freqs)
array([-5., -4., -3., -2., -1., 0., 1., 2., 3., 4.])
```

Shift the zero-frequency component only along the second axis:

```python
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0., 1., 2.],
       [ 3., 4., -4.],
       [-3., -2., -1.]])
```

```python
>>> np.fft.ifftshift(np.fft.fftshift(freqs), axes=(1,))
array([[ 2., 0., 1.],
       [-4., 3., 4.],
       [-1., -3., -2.]])
```

**ifftshift** \((x, \text{axes=\textit{None}})\)

The inverse of \textit{fftshift}.

**Parameters**

- \textit{x} : array_like
  
  Input array.

- \textit{axes} : int or shape tuple, optional
  
  Axes over which to calculate. Defaults to \textit{None}, which shifts all axes.

**Returns**

- \textit{y} : ndarray
  
  The shifted array.

**See Also:**

**fftshift**

Shift zero-frequency component to the center of the spectrum.

**Examples**

```python
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0., 1., 2.],
       [ 3., 4., -4.],
       [-3., -2., -1.]])
```

```python
>>> np.fft.ifftshift(np.fft.fftshift(freqs))
array([[ 0., 1., 2.],
       [ 3., 4., -4.],
       [-3., -2., -1.]])
```
3.6.5 Background information

Fourier analysis is fundamentally a method for expressing a function as a sum of periodic components, and for recovering the signal from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT). The DFT has become a mainstay of numerical computing in part because of a very fast algorithm for computing it, called the Fast Fourier Transform (FFT), which was known to Gauss (1805) and was brought to light in its current form by Cooley and Tukey [CT]. Press et al. [NR] provide an accessible introduction to Fourier analysis and its applications.

Because the discrete Fourier transform separates its input into components that contribute at discrete frequencies, it has a great number of applications in digital signal processing, e.g., for filtering, and in this context the discretized input to the transform is customarily referred to as a signal, which exists in the time domain. The output is called a spectrum or transform and exists in the frequency domain.

There are many ways to define the DFT, varying in the sign of the exponent, normalization, etc. In this implementation, the DFT is defined as

\[
A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i \frac{mk}{n}\right\} \quad k = 0, \ldots, n - 1.
\]

The DFT is in general defined for complex inputs and outputs, and a single-frequency component at linear frequency \(f\) is represented by a complex exponential \(a_m = \exp\left\{2\pi i f m \Delta t\right\}\), where \(\Delta t\) is the sampling interval.

The values in the result follow so-called “standard” order: If \(A = \text{fft}(a, n)\), then \(A[0]\) contains the zero-frequency term (the mean of the signal), which is always purely real for real inputs. Then \(A[1:n/2]\) contains the positive-frequency terms, and \(A[n/2+1:]\) contains the negative-frequency terms, in order of decreasingly negative frequency. For an even number of input points, \(A[n/2]\) represents both positive and negative Nyquist frequency, and is also purely real for real input. For an odd number of input points, \(A[(n-1)/2]\) contains the largest positive frequency, while \(A[(n+1)/2]\) contains the largest negative frequency. The routine \(\text{np.fft.fftfreq}(A)\) returns an array giving the frequencies of corresponding elements in the output. The routine \(\text{np.fft.fftshift}(A)\) shifts transforms and their frequencies to put the zero-frequency components in the middle, and \(\text{np.fft.ifftshift}(A)\) undoes that shift.

When the input \(a\) is a time-domain signal and \(A = \text{fft}(a)\), \(\text{np.abs}(A)\) is its amplitude spectrum and \(\text{np.abs}(A)^*2\) is its power spectrum. The phase spectrum is obtained by \(\text{np.angle}(A)\).

The inverse DFT is defined as

\[
a_m = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp\left\{2\pi i \frac{mk}{n}\right\} \quad n = 0, \ldots, n - 1.
\]

It differs from the forward transform by the sign of the exponential argument and the normalization by \(1/n\).

Real and Hermitian transforms

When the input is purely real, its transform is Hermitian, i.e., the component at frequency \(f_k\) is the complex conjugate of the component at frequency \(-f_k\), which means that for real inputs there is no information in the negative frequency components that is not already available from the positive frequency components. The family of \texttt{rfft} functions is designed to operate on real inputs, and exploits this symmetry by computing only the positive frequency components, up to and including the Nyquist frequency. Thus, \(n\) input points produce \(n/2+1\) complex output points. The inverses of this family assumes the same symmetry of its input, and for an output of \(n\) points uses \(n/2+1\) input points.

Correspondingly, when the spectrum is purely real, the signal is Hermitian. The \texttt{hfft} family of functions exploits this symmetry by using \(n/2+1\) complex points in the input (time) domain for \(n\) real points in the frequency domain.
In higher dimensions, FFTs are used, e.g., for image analysis and filtering. The computational efficiency of the FFT means that it can also be a faster way to compute large convolutions, using the property that a convolution in the time domain is equivalent to a point-by-point multiplication in the frequency domain.

In two dimensions, the DFT is defined as

\[ A_{kl} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{mn} \exp\left\{ -2\pi i \left( \frac{mk}{M} + \frac{nl}{N} \right) \right\} \quad k = 0, \ldots, N - 1; \quad l = 0, \ldots, M - 1, \]

which extends in the obvious way to higher dimensions, and the inverses in higher dimensions also extend in the same way.

References

Examples

For examples, see the various functions.

## 3.7 Linear algebra (numpy.linalg)

### 3.7.1 Matrix and vector products

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot(a, b)</td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td>vdot()</td>
<td>Return the dot product of two vectors.</td>
</tr>
<tr>
<td>inner(a, b)</td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td>outer(a, b)</td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td>tensordot(a, b, axes)</td>
<td>Compute tensor dot product along specified axes for arrays &gt;= 1-D.</td>
</tr>
<tr>
<td>linalg.matrix_power(M, n)</td>
<td>Raise a square matrix to the (integer) power ( n ).</td>
</tr>
<tr>
<td>kron(a, b)</td>
<td>Kronecker product of two arrays.</td>
</tr>
</tbody>
</table>

**dot()**

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For \( N \) dimensions it is a sum product over the last axis of \( a \) and the second-to-last of \( b \):

\[ \text{dot}(a, b)[i,j,k,m] = \text{sum}(a[i,j,:] * b[k,:,m]) \]

**Parameters**

- **a**: array_like
  
  First argument.

- **b**: array_like
  
  Second argument.

**Returns**

- **output**: ndarray
  
  Returns the dot product of \( a \) and \( b \). If \( a \) and \( b \) are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned.
Raises

ValueError:

If the last dimension of a is not the same size as the second-to-last dimension of b.

See Also:

vdot
Complex-conjugating dot product.
tensordot
Sum products over arbitrary axes.

Examples

>>> np.dot(3, 4)
12

Neither argument is complex-conjugated:

>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)

For 2-D arrays it’s the matrix product:

>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])

>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128

vdot()
Return the dot product of two vectors.

The vdot(a, b) function handles complex numbers differently than dot(a, b). If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (with complex conjugation of a). For N dimensions it is a sum product over the last axis of a and the second-to-last of b:

dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

Parameters

a : array_like
    If a is complex the complex conjugate is taken before calculation of the dot product.

b : array_like
    Second argument to the dot product.
**Returns**

output : ndarray

Returns dot product of a and b. Can be an int, float, or complex depending on the types of a and b.

**See Also:**

**dot**

Return the dot product without using the complex conjugate of the first argument.

**Notes**

The dot product is the summation of element wise multiplication.

\[
a \cdot b = \sum_{i=1}^{n} a_i^* b_i = a_1^* b_1 + a_2^* b_2 + \cdots + a_n^* b_n
\]

**Examples**

```python
>>> a = np.array([[1+2j, 3+4j]])
>>> b = np.array([[5+6j, 7+8j]])
>>> np.vdot(a, b)
(70-8j)
>>> np.vdot(b, a)
(70+8j)
>>> a = np.array([[1, 4], [5, 6]])
>>> b = np.array([[4, 1], [2, 2]])
>>> np.vdot(a, b)
30
>>> np.vdot(b, a)
30
```

**inner()**

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

Parameters

a, b : array_like

If a and b are nonscalar, their last dimensions of must match.

Returns

out : ndarray

out.shape = a.shape[:-1] + b.shape[:-1]

Raises

ValueError : If the last dimension of a and b has different size.

See Also:

tensordot

Sum products over arbitrary axes.
**dot**

Generalised matrix product, using second last dimension of \textit{b}.

**Notes**

For vectors (1-D arrays) it computes the ordinary inner-product:

\[
\text{np.inner}(a, b) = \text{sum}(a[^*] \cdot b[^*])
\]

More generally, if \textit{ndim}(a) = \textit{r} > 0 and \textit{ndim}(b) = \textit{s} > 0:

\[
\text{np.inner}(a, b) = \text{np.tensordot}(a, b, \text{axes}=(-1,-1))
\]

or explicitly:

\[
\text{np.inner}(a, b)[i0, ..., ir-1, j0, ..., js-1] = \text{sum}(a[i0, ..., ir-1, :] \cdot b[j0, ..., js-1, :])
\]

In addition \textit{a} or \textit{b} may be scalars, in which case:

\[
\text{np.inner}(a,b) = a \cdot b
\]

**Examples**

Ordinary inner product for vectors:

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([0, 1, 0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[14, 38, 62],
       [86, 110, 134]])
```

An example where \textit{b} is a scalar:

```python
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

**outer** \((a, b)\)

Compute the outer product of two vectors.

Given two vectors, \(a = [a_0, a_1, \ldots, a_M]\) and \(b = [b_0, b_1, \ldots, b_N]\), the outer product \cite{R63} is:

\[
\begin{bmatrix}
[a_0 \cdot b_0 & a_0 \cdot b_1 & \ldots & a_0 \cdot b_N ]
[a_1 \cdot b_0
[ ... 
[a_M \cdot b_0 & a_M \cdot b_N ]
\end{bmatrix}
\]
Parameters

\(a, b\): array_like, shape (M,), (N,)

First and second input vectors. Inputs are flattened if they are not already 1-dimensional.

Returns

out: ndarray, shape (M, N)

\(\text{out}[i, j] = a[i] \times b[j]\)

References

[R63]

Examples

Make a \textit{(very coarse)} grid for computing a Mandelbrot set:

```python
>>> r1 = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> r1
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]])
```

```python
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j,  0.+2.j,  0.+2.j,  0.+2.j,  0.+2.j],
        [ 0.+1.j,  0.+1.j,  0.+1.j,  0.+1.j,  0.+1.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
        [ 0.-1.j,  0.-1.j,  0.-1.j,  0.-1.j,  0.-1.j],
        [ 0.-2.j,  0.-2.j,  0.-2.j,  0.-2.j,  0.-2.j]])
```

```python
>>> grid = r1 + im
>>> grid
array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]])
```

An example using a \textit{“vector”} of letters:

```python
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
       [b, bb, bbb],
       [c, cc, ccc]], dtype=object)
```

\texttt{tensordot} \((a, b, axes=2)\)

Compute tensor dot product along specified axes for arrays \(\geq 1\)-D.

Given two tensors (arrays of dimension greater than or equal to one), \(a\) and \(b\), and an array_like object containing two array_like objects, \((a\_axes, b\_axes)\), sum the products of \(a\)'s and \(b\)'s elements (components) over the axes specified by \(a\_axes\) and \(b\_axes\). The third argument can be a single non-negative integer_like scalar, \(N\); if it is such, then the last \(N\) dimensions of \(a\) and the first \(N\) dimensions of \(b\) are summed over.

Parameters

\(a, b\): array_like, len(shape) \(\geq 1\)

Tensors to “dot”.  

---

3.7. Linear algebra (\texttt{numpy.linalg})
axes: variable type

* integer_like scalar:
  Number of axes to sum over (applies to both arrays); or

* array_like, shape = (2,), both elements array_like:
  Axes to be summed over, first sequence applying to a, second to b.

See Also:
numpy.dot

Notes

When there is more than one axis to sum over - and they are not the last (first) axes of a (b) - the argument axes should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

Examples

A “traditional” example:

```python
>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> c = np.tensordot(a,b, axes=(1,0), (0,1))
>>> c.shape
(5, 2)
>>> c
array([[ 4400., 4730.],
       [ 4532., 4874.],
       [ 4664., 5018.],
       [ 4796., 5162.],
       [ 4928., 5306.]]);
>>> # A slower but equivalent way of computing the same...
>>> d = np.zeros((5,2))
>>> for i in range(5):
...     for j in range(2):
...         for k in range(3):
...             for n in range(4):
...                 d[i,j] += a[k,n,i] * b[n,k,j]
>>> c == d
array([[ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True]], dtype=bool)
```

An extended example taking advantage of the overloading of + and *:

```python
>>> a = np.array(range(1, 9))
>>> a.shape = (2, 2, 2)
>>> A = np.array(('a', 'b', 'c', 'd'), dtype=object)
>>> A.shape = (2, 2)
>>> a; A
array([[[1, 2],
         [3, 4]],
       [[5, 6],
         [7, 8]]])
```
array([[a, b],
       [c, d]], dtype=object)

>>> np.tensordot(a, A, 0)  # "Left for reader" (result too long to incl.)

>>> np.tensordot(a, A, (0, 1))
array([[abb, cdd],
       [aaabbb, cccddd]], dtype=object)

array([[0, 1],
       [0, 1]])
array([[abb, cdd],
       [aaabbb, cccddd]], dtype=object)

matrix_power (M, n)
Raise a square matrix to the (integer) power n.
For positive integers n, the power is computed by repeated matrix
squarings and matrix multiplications. If n == 0, the identity matrix
of the same shape as M is returned. If n < 0, the inverse is computed
and then raised to the abs(n).

Parameters
M : ndarray or matrix object
    Matrix to be “powered.” Must be square, i.e. M.shape == (m, m),
    with m a positive integer.

n : int
    The exponent can be any integer or long integer, positive, negative,
    or zero.

Returns
M**n : ndarray or matrix object
    The return value is the same shape and type as M; if the exponent
    is positive or zero then the type of the elements is the same as those
    of M. If the exponent is negative the elements are floating-point.
Raises

LinAlgError :

If the matrix is not numerically invertible.

See Also:

matrix

Provides an equivalent function as the exponentiation operator (**, not ^).

Examples

```python
>>> from numpy import linalg as LA
>>> i = np.array([[0, 1], [-1, 0]]) # matrix equiv. of the imaginary unit
>>> LA.matrix_power(i, 3) # should = -i
array([[ 0, -1],
       [ 1,  0]])
>>> LA.matrix_power(np.matrix(i), 3) # matrix arg returns matrix
matrix([[ 0, -1],
        [ 1,  0]])
>>> LA.matrix_power(i, 0)
array([[ 1,  0],
       [ 0,  1]])
>>> LA.matrix_power(i, -3) # should = 1/(-i) = i, but w/ f.p. elements
array([[ 0., 1.],
        [-1., 0.]]
```

Somewhat more sophisticated example

```python
>>> q = np.zeros((4, 4))
>>> q[0:2, 0:2] = -i
>>> q[2:4, 2:4] = i
>>> q # one of the three quarternion units not equal to 1
array([[ 0., -1., 0., 0.],
       [ 1.,  0., 0., 0.],
       [ 0.,  0., 0., 1.],
       [ 0.,  0., -1., 0.]])
>>> LA.matrix_power(q, 2) # = -np.eye(4)
array([[-1., 0., 0., 0.],
       [ 0., -1., 0., 0.],
       [ 0.,  0., -1., 0.],
       [ 0.,  0.,  0., -1.]]
```

\texttt{kron}(a, b)

Kronecker product of two arrays.

Computes the Kronecker product, a composite array made of blocks of the second array scaled by the first.

Parameters

\texttt{a, b} : array_like

Returns

\texttt{out} : ndarray

See Also:

\texttt{outer}

The outer product
Notes

The function assumes that the number of dimensions of \( a \) and \( b \) are the same, if necessary prepending the smallest with ones. If \( a.shape = (r_0, r_1, \ldots, r_N) \) and \( b.shape = (s_0, s_1, \ldots, s_N) \), the Kronecker product has shape \((r_0*s_0, r_1*s_1, \ldots, r_N*s_N)\). The elements are products of elements from \( a \) and \( b \), organized explicitly by:

\[
kron(a, b)[k_0, k_1, \ldots, k_N] = a[i_0, i_1, \ldots, i_N] \times b[j_0, j_1, \ldots, j_N]
\]

where:

\[
kt = it \times st + jt, \quad t = 0, \ldots, N
\]

In the common 2-D case \((N=1)\), the block structure can be visualized:

\[
[ [ a[0,0]*b, \quad a[0,1]*b, \quad \ldots, \quad a[0,-1]*b ],
  [ \quad \ldots \quad \ldots \quad \ldots ],
  [ a[-1,0]*b, \quad a[-1,1]*b, \quad \ldots, \quad a[-1,-1]*b ]]
\]

Examples

```python
def np.kron([1,10,100], [5,6,7])
array([5, 6, 7, 50, 60, 70, 500, 600, 700])
```

```python
def np.kron([5,6,7], [1,10,100])
array([5, 50, 500, 6, 60, 600, 7, 70, 700])
```

```python
def np.kron(np.eye(2), np.ones((2,2)))
array([[1., 1., 0., 0.],[1., 1., 0., 0.],[0., 0., 1., 1.],[0., 0., 1., 1.]])
```

```python
def a = np.arange(100).reshape((2,5,2,5))
def b = np.arange(24).reshape((2,3,4))
def c = np.kron(a,b)
def c.shape
(2, 10, 6, 20)
def I = (1,3,0,2)
def J = (0,2,1)
def J1 = (0,) + J  # extend to ndim=4
def S1 = (1,) + b.shape
def K = tuple(np.array(I) * np.array(S1) + np.array(J1))
True
```

3.7.2 Decompositions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>linalg.cholesky(a)</code></td>
<td>Cholesky decomposition.</td>
</tr>
<tr>
<td><code>linalg.qr (a[, mode])</code></td>
<td>Compute the qr factorization of a matrix.</td>
</tr>
<tr>
<td><code>linalg.svd (a[, full_matrices, compute_uv])</code></td>
<td>Singular Value Decomposition.</td>
</tr>
</tbody>
</table>

cholesky \((a)\)

Cholesky decomposition.
Return the Cholesky decomposition, \( L * L.H \), of the square matrix \( a \), where \( L \) is lower-triangular and \( L.H \) is the conjugate transpose operator (which is the ordinary transpose if \( a \) is real-valued). \( a \) must be Hermitian (symmetric if real-valued) and positive-definite. Only \( L \) is actually returned.

Parameters
- \( a \) : array_like, shape (M, M)
  Hermitian (symmetric if all elements are real), positive-definite input matrix.

Returns
- \( L \) : ndarray, or matrix object if \( a \) is, shape (M, M)
  Lower-triangular Cholesky factor of \( a \).

Raises
- LinAlgError : If the decomposition fails, for example, if \( a \) is not positive-definite.

Notes
The Cholesky decomposition is often used as a fast way of solving
\[
A x = b
\]
when \( A \) is both Hermitian/symmetric and positive-definite).
First, we solve for \( y \) in
\[
L y = b,
\]
and then for \( x \) in
\[
L.H x = y.
\]

Examples
>>> A = np.array([[1,-2j],[2j,5]])
>>> A
array([[ 1.+0.j, 0.-2.j],
[ 0.+2.j, 5.+0.j]])
>>> L = np.linalg.cholesky(A)
>>> L
array([[ 1.+0.j, 0.+0.j],
[ 0.+2.j, 1.+0.j]])
>>> np.dot(L, L.T.conj())  # verify that L * L.H = A
array([[ 1.+0.j, 0.-2.j],
[ 0.+2.j, 5.+0.j]])
>>> A = [[1,-2j],[2j,5]]  # what happens if A is only array_like?
>>> np.linalg.cholesky(A)  # an ndarray object is returned
array([[ 1.+0.j, 0.+0.j],
[ 0.+2.j, 1.+0.j]])
>>> # But a matrix object is returned if A is a matrix object
>>> LA.cholesky(np.matrix(A))
matrix([[ 1.+0.j, 0.+0.j],
[ 0.+2.j, 1.+0.j]])
Compute the qr factorization of a matrix.

Factor the matrix \(a\) as \(qr\), where \(q\) is orthonormal (\(\text{dot}(q_{i,:},q_{i,:}) = \delta_{ij}\), the Kronecker delta) and \(r\) is upper-triangular.

**Parameters**

- **a**: array_like, shape (M, N)
  - Matrix to be factored.
- **mode**: {'full', 'r', 'economic'}
  - Specifies the information to be returned. 'full' is the default. mode='r' returns a “true” \(r\), while 'economic' returns a “polluted” \(r\) (albeit slightly faster; see Returns below).

**Returns**

* If mode = ‘full’:
  - \(q\) : ndarray of float or complex, shape (M, K)
  - \(r\) : ndarray of float or complex, shape (K, N)
  - Size \(K = \min(M, N)\)
* If mode = ‘r’:
  - \(r\) : ndarray of float or complex, shape (K, N)
* If mode = ‘economic’:
  - \(a2\) : ndarray of float or complex, shape (M, N)
  - The diagonal and the upper triangle of \(a2\) contains \(r\), while the rest of the matrix is undefined.

**Raises**

- LinAlgError
  - If factoring fails.

**Notes**

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dorgqr, and zungqr.

For more information on the qr factorization, see for example: [http://en.wikipedia.org/wiki/QR_factorization](http://en.wikipedia.org/wiki/QR_factorization)

Subclasses of `ndarray` are preserved, so if \(a\) is of type `matrix`, all the return values will be matrices too.

**Examples**

```python
>>> a = np.random.randn(9, 6)
>>> q, r = np.linalg.qr(a)
>>> np.allclose(a, np.dot(q, r))  # a does equal qr
True
>>> r2 = np.linalg.qr(a, mode='r')
>>> r3 = np.linalg.qr(a, mode='economic')
>>> np.allclose(r, r2)  # mode='r' returns the same r as mode='full'
True
>>> np.allclose(r, np.triu(r3[:6,:6], k=0))  # But only triu parts are guaranteed equal when mode='economic'
True
```
Example illustrating a common use of qr: solving of least squares problems

What are the least-squares-best \( m \) and \( y0 \) in \( y = y0 + mx \) for the following data: \{(0,1), (1,0), (1,2), (2,1)\}. (Graph the points and you’ll see that it should be \( y0 = 0, m = 1 \).) The answer is provided by solving the over-determined matrix equation \( Ax = b \), where:

\[
A = \begin{bmatrix}
[0, 1], [1, 1], [1, 1], [2, 1]
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
y0, m\n\end{bmatrix}
\]

\[
b = \begin{bmatrix}
[1], [0], [2], [1]
\end{bmatrix}
\]

If \( A = qr \) such that \( q \) is orthonormal (which is always possible via Gram-Schmidt), then \( x = \text{inv}(r) \ast (q.T) \ast b \). (In numpy practice, however, we simply use \text{lqstsq}.)

```python
>>> A = np.array([[0, 1], [1, 1], [1, 1], [2, 1]])
>>> A
array([[0, 1],
[1, 1],
[1, 1],
[2, 1]])
>>> b = np.array([1, 0, 2, 1])
>>> q, r = LA.qr(A)
>>> p = np.dot(q.T, b)
>>> np.dot(LA.inv(r), p)
array([1.1e-16, 1.0e+00])
```

**svd** (\( a, \text{full\_matrices}=1, \text{compute\_uv}=1 \))

Singular Value Decomposition.

Factors the matrix \( a \) into \( u \ast \text{np.diag}(s) \ast v.H \), where \( u \) and \( v \) are unitary (i.e., \( u.H = \text{inv}(u) \) and similarly for \( v \)). \( .H \) is the conjugate transpose operator (which is the ordinary transpose for real-valued matrices), and \( s \) is a 1-D array of \( a \)'s singular values.

**Parameters**

- **a**: array_like

  Matrix of shape \((M, N)\) to decompose.

- **full_matrices**: bool, optional

  If True (default), \( u \) and \( v.H \) have the shapes \((M, M)\) and \((N, N)\), respectively. Otherwise, the shapes are \((M, K)\) and \((K, N)\), resp., where \( K = \min(M, N) \).

- **compute_uv**: bool, optional

  Whether or not to compute \( u \) and \( v.H \) in addition to \( s \). True by default.

**Returns**

- **u**: ndarray

  Unitary matrix. The shape of \( U \) is \((M, M)\) or \((M, K)\) depending on value of \text{full_matrices}.

- **s**: ndarray

  The singular values, sorted so that \( s[i] >= s[i+1] \). \( S \) is a 1-D array of length \( \min(M, N) \).

- **v.H**: ndarray

  Unitary matrix of shape \((N, N)\) or \((K, N)\), depending on \text{full_matrices}.

**Raises**

- **LinAlgError**: 

---
If SVD computation does not converge.

Notes

If \( a \) is a matrix object (as opposed to an \texttt{ndarray} ), then so are all the return values.

Examples

```python
>>> a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
>>> U, s, Vh = np.linalg.svd(a)
>>> U.shape, Vh.shape, s.shape
((9, 9), (6, 6), (6,))

>>> U, s, Vh = np.linalg.svd(a, full_matrices=False)
>>> U.shape, Vh.shape, s.shape
((9, 6), (6, 6), (6,))

>>> S = np.diag(s)
>>> np.allclose(a, np.dot(U, np.dot(S, Vh)))
True

>>> s2 = np.linalg.svd(a, compute_uv=False)
>>> np.allclose(s, s2)
True
```

### 3.7.3 Matrix eigenvalues

```
linalg.eig(a)        Compute the eigenvalues and right eigenvectors of a square array.
linalg.eigh(a[, UPLO]) Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.
linalg.eigvals(a)    Compute the eigenvalues of a general matrix.
linalg.eigvalsh(a[, UPLO]) Compute the eigenvalues of a Hermitian or real symmetric matrix.
eig(a)               Compute the eigenvalues and right eigenvectors of a square array.
```

Compute the eigenvalues and right eigenvectors of a square array.

**Parameters**

- \( a \) : array_like, shape (M, M)
  - A square array of real or complex elements.

**Returns**

- \( w \) : ndarray, shape (M,)
  - The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered, nor are they necessarily real for real arrays (though for real arrays complex-valued eigenvalues should occur in conjugate pairs).

- \( v \) : ndarray, shape (M, M)
  - The normalized (unit “length”) eigenvectors, such that the column \( v[:,i] \) is the eigenvector corresponding to the eigenvalue \( w[i] \).

**Raises**

- \texttt{LinAlgError}:
  - If the eigenvalue computation does not converge.

**See Also**
**eigvals**
eigenvalues of a non-symmetric array.

**Notes**
This is a simple interface to the LAPACK routines dgeev and zgeev which compute the eigenvalues and eigenvectors of, respectively, general real- and complex-valued square arrays.

The number \( w \) is an eigenvalue of \( a \) if there exists a vector \( v \) such that \( \text{dot}(a, v) = w \cdot v \). Thus, the arrays \( a, w, \) and \( v \) satisfy the equations \( \text{dot}(a[i,:], v[i]) = w[i] \cdot v[:,i] \) for \( i \in \{0, ..., M - 1\} \).

The array \( v \) of eigenvectors may not be of maximum rank, that is, some of the columns may be linearly dependent, although round-off error may obscure that fact. If the eigenvalues are all different, then theoretically the eigenvectors are linearly independent. Likewise, the (complex-valued) matrix of eigenvectors \( v \) is unitary if the matrix \( a \) is normal, i.e., if \( \text{dot}(a, a.H) = \text{dot}(a.H, a) \), where \( a.H \) denotes the conjugate transpose of \( a \).

Finally, it is emphasized that \( v \) consists of the right (as in right-hand side) eigenvectors of \( a \). A vector \( y \) satisfying \( \text{dot}(y.T, a) = z \cdot y.T \) for some number \( z \) is called a left eigenvector of \( a \), and, in general, the left and right eigenvectors of a matrix are not necessarily the (perhaps conjugate) transposes of each other.

**References**

**Examples**
>>> from numpy import linalg as LA

(Almost) trivial example with real e-values and e-vectors.

>>> w, v = LA.eig(np.diag((1, 2, 3)))
>>> w; v
array([ 1., 2., 3.])
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])

Real matrix possessing complex e-values and e-vectors; note that the e-values are complex conjugates of each other.

>>> w, v = LA.eig(np.array([[1, -1], [1, 1]]))
>>> w; v
array([ 1. + 1.j,  1. - 1.j])
array([[ 0.70710678+0.j ,  0.70710678+0.j   ],
       [ 0.00000000-0.70710678j, 0.00000000+0.70710678j]])

Complex-valued matrix with real e-values (but complex-valued e-vectors); note that \( a.\text{conj()}.T = a \), i.e., \( a \) is Hermitian.

>>> a = np.array([[1, 1j], [-1j, 1]])
>>> w, v = LA.eig(a)
>>> w; v
array([ 2.00000000e+00+0.j,  5.98651912e-36+0.j]) # i.e., (2, 0)
array([[0.00000000+0.70710678j, 0.70710678+0.j],
       [0.70710678+0.j, 0.00000000+0.70710678j]])

Be careful about round-off error!

```python
>>> a = np.array([[1 + 1e-9, 0], [0, 1 - 1e-9]])
>>> # Theor. e-values are 1 +/- 1e-9
>>> w, v = LA.eig(a)
>>> w; v
array([ 1., 1.])
array([[ 1., 0.],
       [ 0., 1.]])
```

eigh \( (a, \text{UPLO}=\text{L}') \)

Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of \( a \), and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

**Parameters**

- \( a \) : array_like, shape (M, M)
  - A complex Hermitian or real symmetric matrix.
- \( \text{UPLO} \) : {'L', 'U'}, optional
  - Specifies whether the calculation is done with the lower triangular part of \( a \) ('L', default) or the upper triangular part ('U').

**Returns**

- \( w \) : ndarray, shape (M,)
  - The eigenvalues, not necessarily ordered.
- \( v \) : ndarray, or matrix object if \( a \) is, shape (M, M)
  - The column \( v[:, i] \) is the normalized eigenvector corresponding to the eigenvalue \( w[i] \).

**Raises**

- LinAlgError : If the eigenvalue computation does not converge.

**See Also:**

- **eigvalsh**
  - eigenvalues of symmetric or Hermitian arrays.
- **eig**
  - eigenvalues and right eigenvectors for non-symmetric arrays.
- **eigvals**
  - eigenvalues of non-symmetric arrays.

**Notes**

This is a simple interface to the LAPACK routines dsyevd and zheevd, which compute the eigenvalues and eigenvectors of real symmetric and complex Hermitian arrays, respectively.

The eigenvalues of real symmetric or complex Hermitian matrices are always real. [R47] The array \( v \) of (column) eigenvectors is unitary and \( a, w, \) and \( v \) satisfy the equations \( \text{dot}(a, v[:, i]) = w[i] \times v[:, i] \).
Examples

```python
>>> from numpy import linalg as LA
>>> a = np.array([[1, -2j], [2j, 5]])
>>> a
array([[ 1.+0.j, 0.-2.j],
       [ 0.+2.j, 5.+0.j]])
>>> w, v = LA.eigh(a)
>>> w; v
array([ 0.17157288, 5.82842712])
array([[-0.92387953+0.j , -0.38268343+0.j ],
       [ 0.00000000+0.38268343j, 0.00000000-0.92387953j]])

>>> np.dot(a, v[:, 0]) - w[0] * v[:, 0]  # verify 1st e-val/vec pair
array([ 2.77555756e-17 +0.j, 0. +1.38777878e-16j])
>>> np.dot(a, v[:, 1]) - w[1] * v[:, 1]  # verify 2nd e-val/vec pair
array([ 0.+0.j, 0.+0.j])

>>> A = np.matrix(a)  # what happens if input is a matrix object
>>> A
matrix([[ 1.+0.j, 0.-2.j],
         [ 0.+2.j, 5.+0.j]])
>>> w, v = LA.eigh(A)
>>> w; v
array([ 0.17157288, 5.82842712])
matrix([[[-0.92387953+0.j , -0.38268343+0.j ],
        [ 0.00000000+0.38268343j, 0.00000000-0.92387953j]])
```

eigvals(a)
Compute the eigenvalues of a general matrix.

Main difference between eigvals and eig: the eigenvectors aren’t returned.

**Parameters**
- `a`: array_like, shape (M, M)
  
  A complex- or real-valued matrix whose eigenvalues will be computed.

**Returns**
- `w`: ndarray, shape (M,)
  
  The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

**Raises**
- `LinAlgError`
  
  If the eigenvalue computation does not converge.

**See Also:**
- `eig`
  
  eigenvalues and right eigenvectors of general arrays
- `eigvalsh`
  
  eigenvalues of symmetric or Hermitian arrays.
**eigh**

eigenvalues and eigenvectors of symmetric/Hermitian arrays.

**Notes**

This is a simple interface to the LAPACK routines dgeev and zgeev that sets those routines’ flags to return only the eigenvalues of general real and complex arrays, respectively.

**Examples**

Illustration, using the fact that the eigenvalues of a diagonal matrix are its diagonal elements, that multiplying a matrix on the left by an orthogonal matrix, $Q$, and on the right by $Q^T$ (the transpose of $Q$), preserves the eigenvalues of the “middle” matrix. In other words, if $Q$ is orthogonal, then $Q * A * Q^T$ has the same eigenvalues as $A$:

```python
>>> from numpy import linalg as LA
>>> x = np.random.random()
>>> Q = np.array([[np.cos(x), -np.sin(x)], [np.sin(x), np.cos(x)]])
>>> LA.norm(Q[0, :]), LA.norm(Q[1, :]), np.dot(Q[0, :], Q[1, :])
(1.0, 1.0, 0.0)
```

Now multiply a diagonal matrix by $Q$ on one side and by $Q^T$ on the other:

```python
>>> D = np.diag((-1,1))
>>> LA.eigvals(D)
array([-1., 1.])
>>> A = np.dot(Q, D)
>>> A = np.dot(A, Q.T)
>>> LA.eigvals(A)
array([ 1., -1.])
```

**eigvalsh** *(a, UPLO='L')*

Compute the eigenvalues of a Hermitian or real symmetric matrix.

Main difference from eigh: the eigenvectors are not computed.

**Parameters**

- **a**: array_like, shape (M, M)
  
  A complex- or real-valued matrix whose eigenvalues are to be computed.

- **UPLO**: {'L', 'U'}, optional
  
  Specifies whether the calculation is done with the lower triangular part of $a$ ('L', default) or the upper triangular part ('U').

**Returns**

- **w**: ndarray, shape (M,)
  
  The eigenvalues, not necessarily ordered, each repeated according to its multiplicity.

**Raises**

- **LinAlgError**:
  
  If the eigenvalue computation does not converge.

**See Also**:

- **eigh**
  
  eigenvalues and eigenvectors of symmetric/Hermitian arrays.
**eigvals**

eigenvalues of general real or complex arrays.

**eig**
eigenvalues and right eigenvectors of general real or complex arrays.

**Notes**

This is a simple interface to the LAPACK routines dsyevd and zheevd that sets those routines’ flags to return only the eigenvalues of real symmetric and complex Hermitian arrays, respectively.

**Examples**

```python
>>> from numpy import linalg as LA
>>> a = np.array([[1, -2j], [2j, 5]])
>>> LA.eigvalsh(a)
array([ 0.17157288+0.j, 5.82842712+0.j])
```

### 3.7.4 Norms and other numbers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linalg.norm(x, ord=None)</td>
<td>Matrix or vector norm.</td>
</tr>
<tr>
<td>linalg.cond(x, p)</td>
<td>Compute the condition number of a matrix.</td>
</tr>
<tr>
<td>linalg.det(a)</td>
<td>Compute the determinant of an array.</td>
</tr>
<tr>
<td>trace(a[, offset, axis1, axis2, ...])</td>
<td>Return the sum along diagonals of the array.</td>
</tr>
</tbody>
</table>

**norm** *(x, ord=’fro’)*

Matrix or vector norm.

This function is able to return one of seven different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

**Parameters**

- **x**: array_like, shape (M,) or (M, N)
  
  Input array.

- **ord**: {non-zero int, inf, -inf, ‘fro’}, optional
  
  Order of the norm (see table under Notes). inf means numpy’s inf object.

**Returns**

- **n**: float
  
  Norm of the matrix or vector.

**Notes**

For values of `ord <= 0`, the result is, strictly speaking, not a mathematical ‘norm’, but it may still be useful for various numerical purposes.

The following norms can be calculated:
### norm for matrices

<table>
<thead>
<tr>
<th>ord</th>
<th>norm for matrices</th>
<th>norm for vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Frobenius norm</td>
<td>2-norm</td>
</tr>
<tr>
<td>‘fro’</td>
<td>Frobenius norm</td>
<td>–</td>
</tr>
<tr>
<td>inf</td>
<td>max(sum(abs(x), axis=1))</td>
<td>max(abs(x))</td>
</tr>
<tr>
<td>-inf</td>
<td>min(sum(abs(x), axis=1))</td>
<td>min(abs(x))</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>sum(x != 0)</td>
</tr>
<tr>
<td>1</td>
<td>max(sum(abs(x), axis=0))</td>
<td>as below</td>
</tr>
<tr>
<td>-1</td>
<td>min(sum(abs(x), axis=0))</td>
<td>as below</td>
</tr>
<tr>
<td>2</td>
<td>2-norm (largest sing. value)</td>
<td>as below</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>other</td>
<td>–</td>
<td>sum(abs(x)<strong>ord)</strong>(1./ord)</td>
</tr>
</tbody>
</table>

The Frobenius norm is given by [R48]:

\[ ||A||_F = \left(\sum_{i,j} |a_{i,j}|^2\right)^{1/2} \]

### References

[R48]

### Examples

```python
>>> from numpy import linalg as LA
>>> a = np.arange(9) - 4
>>> a
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> b = a.reshape((3, 3))
>>> b
array([[-4, -3, -2],
       [-1,  0,  1],
       [ 2,  3,  4]])
```

```python
>>> LA.norm(a)
7.745966692414834
>>> LA.norm(b)
7.745966692414834
>>> LA.norm(b, 'fro')
7.745966692414834
>>> LA.norm(a, np.inf)
4
>>> LA.norm(b, np.inf)
9
>>> LA.norm(a, -np.inf)
0
>>> LA.norm(b, -np.inf)
2

>>> LA.norm(a, 1)
20
>>> LA.norm(b, 1)
7
>>> LA.norm(a, -1)
-4.656128774142013e-010
>>> LA.norm(b, -1)
6
>>> LA.norm(a, 2)
7.745966692414834
```
>>> LA.norm(b, 2)
7.3484692283495345

>>> LA.norm(a, -2)
nan

>>> LA.norm(b, -2)
1.8570313885190563e-016

>>> LA.norm(a, 3)
5.848034764257312

>>> LA.norm(a, -3)
nan

cond (x, p=None)
Compute the condition number of a matrix.
This function is capable of returning the condition number using one of seven different norms, depending on the value of p (see Parameters below).

Parameters
  x : array_like, shape (M, N)
      The matrix whose condition number is sought.
  p : {None, 1, -1, 2, -2, inf, -inf, 'fro'}, optional
      Order of the norm:

<table>
<thead>
<tr>
<th>p</th>
<th>norm for matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>2-norm, computed directly using the SVD</td>
</tr>
<tr>
<td>'fro'</td>
<td>Frobenius norm</td>
</tr>
<tr>
<td>inf</td>
<td>max(sum(abs(x), axis=1))</td>
</tr>
<tr>
<td>-inf</td>
<td>min(sum(abs(x), axis=1))</td>
</tr>
<tr>
<td>1</td>
<td>max(sum(abs(x), axis=0))</td>
</tr>
<tr>
<td>-1</td>
<td>min(sum(abs(x), axis=0))</td>
</tr>
<tr>
<td>2</td>
<td>2-norm (largest sing. value)</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
</tr>
</tbody>
</table>

inf means the numpy.inf object, and the Frobenius norm is the root-of-sum-of-squares norm.

Returns
c : {float, inf}
    The condition number of the matrix. May be infinite.

See Also:
numpy.linalg.linalg.norm

Notes
The condition number of x is defined as the norm of x times the norm of the inverse of x [R46]; the norm can be the usual L2-norm (root-of-sum-of-squares) or one of a number of other matrix norms.

References
[R46]
Examples

```python
>>> from numpy import linalg as LA
>>> a = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1]])
>>> a
array([[ 1, 0, -1],
       [ 0, 1, 0],
       [ 1, 0, 1]])
>>> LA.cond(a)
1.4142135623730951
>>> LA.cond(a, 'fro')
3.1622776601683795
>>> LA.cond(a, np.inf)
2.0
>>> LA.cond(a, -np.inf)
1.0
>>> LA.cond(a, 1)
2.0
>>> LA.cond(a, -1)
1.0
>>> LA.cond(a, 2)
1.4142135623730951
>>> LA.cond(a, -2)
0.70710678118654746
>>> min(LA.svd(a, compute_uv=0))*min(LA.svd(LA.inv(a), compute_uv=0))
0.70710678118654746
```

`det (a)`
Compute the determinant of an array.

**Parameters**
- `a`: array_like, shape (M, M)
  
  Input array.

**Returns**
- `det`: ndarray
  
  Determinant of `a`.

**Notes**

The determinant is computed via LU factorization using the LAPACK routine z/dgetrf.

**Examples**

The determinant of a 2-D array \([a, b], [c, d]\) is \(ad - bc\):

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0
```

`trace (a, offset=0, axis1=0, axis2=1, dtype=None, out=None)`
Return the sum along diagonals of the array.

If `a` is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements `a[i, i+offset]` for all `i`.

If `a` has more than two dimensions, then the axes specified by `axis1` and `axis2` are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of `a` with `axis1` and `axis2` removed.
Parameters

a : array_like
   Input array, from which the diagonals are taken.

offset : int, optional
   Offset of the diagonal from the main diagonal. Can be both positive and negative.
   Defaults to 0.

axis1, axis2 : int, optional
   Axes to be used as the first and second axis of the 2-D sub-arrays from which the
   diagonals should be taken. Defaults are the first two axes of a.

dtype : dtype, optional
   Determines the data-type of the returned array and of the accumulator where the ele-
   ments are summed. If dtype has the value None and a is of integer type of precision less
   than the default integer precision, then the default integer precision is used. Otherwise,
   the precision is the same as that of a.

out : ndarray, optional
   Array into which the output is placed. Its type is preserved and it must be of the right
   shape to hold the output.

Returns

sum_along_diagonals : ndarray
   If a is 2-D, the sum along the diagonal is returned. If a has larger dimensions, then an
   array of sums along diagonals is returned.

See Also:
diag, diagonal, diagflat

Examples

>>> np.trace(np.eye(3))
3.0
>>> a = np.arange(8).reshape((2,2,2))
>>> np.trace(a)
array([6, 8])

>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)

3.7.5 Solving equations and inverting matrices

| linalg.solve(a, b) | Solve a linear matrix equation, or system of linear scalar equations. |
| linalg.tensorsolve(a, b[, axes]) | Solve the tensor equation \( a \times = b \) for \( x \). |
| linalg.lstsq(a, b[, rcond]) | Return the least-squares solution to a linear matrix equation. |
| linalg.inv(a) | Compute the (multiplicative) inverse of a matrix. |
| linalg.pinv(a[, rcond]) | Compute the (Moore-Penrose) pseudo-inverse of a matrix. |
| linalg.tensorinv(a[, ind]) | Compute the ‘inverse’ of an N-dimensional array. |
**solve**(*a*, *b*)

Solve a linear matrix equation, or system of linear scalar equations.

Computes the “exact” solution, *x*, of the well-determined, i.e., full rank, linear matrix equation *a* *x* = *b*.

**Parameters**

- **a**: array_like, shape (M, M)
  - Coefficient matrix.
- **b**: array_like, shape (M,) or (M, N)
  - Ordinate or “dependent variable” values.

**Returns**

- **x**: ndarray, shape (M,) or (M, N) depending on *b*
  - Solution to the system a *x* = *b*

**Raises**

- **LinAlgError**:
  - If *a* is singular or not square.

**Notes**

`solve` is a wrapper for the LAPACK routines `dgesv` and `zgesv`, the former being used if *a* is real-valued, the latter if it is complex-valued. The solution to the system of linear equations is computed using an LU decomposition [R50] with partial pivoting and row interchanges.

*a* must be square and of full-rank, i.e., all rows (or, equivalently, columns) must be linearly independent; if either is not true, use `lstsq` for the least-squares best “solution” of the system/equation.

**References**

[R50]

**Examples**

Solve the system of equations 3 * x0 + x1 = 9 and x0 + 2 * x1 = 8:

```python
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

Check that the solution is correct:

```python
>>> (np.dot(a, x) == b).all()
True
```

**tensorsolve**(*a*, *b*, *axes=None*)

Solve the tensor equation *a* *x* = *b* for *x*.

It is assumed that all indices of *x* are summed over in the product, together with the rightmost indices of *a*, as is done in, for example, `tensordot(a, x, axes=len(b.shape))`.

**Parameters**

- **a**: array_like
Coefficient tensor, of shape \( b . \text{shape} + Q \), a tuple, equals the shape of that sub-tensor of \( a \) consisting of the appropriate number of its rightmost indices, and must be such that

\[
\prod(Q) == \prod(b.\text{shape}) \quad \text{(in which sense} \ a \ \text{is said to be ‘square’)}.
\]

\( b : \text{array_like} \)
Right-hand tensor, which can be of any shape.

\( \text{axes} : \text{tuple of ints, optional} \)
Axes in \( a \) to reorder to the right, before inversion. If None (default), no reordering is done.

Returns
\( x : \text{ndarray, shape} \ Q \)

Raises
LinAlgError :
If \( a \) is singular or not ‘square’ (in the above sense).

See Also:
tensordot, tensorinv

Examples

\[
\begin{align*}
\text{>>> } & a = \text{np.eye}(2*3*4) \\
\text{>>> } & a.\text{shape} = (2*3, 4, 2, 3, 4) \\
\text{>>> } & b = \text{np.random.randn}(2*3, 4) \\
\text{>>> } & x = \text{np.linalg.tensorsolve}(a, b) \\
\text{>>> } & x.\text{shape} \\
& (2, 3, 4) \\
\text{>>> } & \text{np.allclose(np.tensordot(a, x, axes=3), b)} \\
& \text{True}
\end{align*}
\]

\text{linalg.solve}(a, \ \text{b, rcond=-1})
Return the least-squares solution to a linear matrix equation.

Solves the equation \( a x = b \) by computing a vector \( x \) that minimizes the norm \( \| b - a x \| \). The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of \( a \) can be less than, equal to, or greater than its number of linearly independent columns). If \( a \) is square and of full rank, then \( x \) (but for round-off error) is the “exact” solution of the equation.

Parameters
\( a : \text{array_like, shape} (M, N) \)
“Coefficient” matrix.

\( b : \text{array_like, shape} (M,) \ or \ (M, K) \)
Ordinate or “dependent variable” values. If \( b \) is two-dimensional, the least-squares solution is calculated for each of the \( K \) columns of \( b \).

\( \text{rcond} : \text{float, optional} \)
Cut-off ratio for small singular values of \( a \). Singular values are set to zero if they are smaller than \( rcond \times \) the largest singular value of \( a \).

Returns
\( x : \text{ndarray, shape} (N,) \ or \ (N, K) \)
Least-squares solution. The shape of $x$ depends on the shape of $b$.

- **residues** : ndarray, shape (), (1,), or (K,)
  Sums of residues; squared Euclidean norm for each column in $b - a \cdot x$. If the rank of $a$ is $< N$ or $> M$, this is an empty array. If $b$ is 1-dimensional, this is a (1,) shape array. Otherwise the shape is (K,).

- **rank** : int
  Rank of matrix $a$.

- **s** : ndarray, shape (min(M,N),)
  Singular values of $a$.

**Raises**

- **LinAlgError** :
  If computation does not converge.

**Notes**

If $b$ is a matrix, then all array results are returned as matrices.

**Examples**

Fit a line, $y = mx + c$, through some noisy data-points:

```python
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])
```

By examining the coefficients, we see that the line should have a gradient of roughly 1 and cut the y-axis at, more or less, -1.

We can rewrite the line equation as $y = Ap$, where $A = \begin{bmatrix} x & 1 \end{bmatrix}$ and $p = \begin{bmatrix} m \\ c \end{bmatrix}$. Now use `linalg.lstsq` to solve for $p$:

```python
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.]])
```

```python
>>> m, c = np.linalg.lstsq(A, y)[0]
>>> print m, c
1.0 -0.95
```

Plot the data along with the fitted line:

```python
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o', label='Original data', markersize=10)
>>> plt.plot(x, m*x + c, 'r', label='Fitted line')
>>> plt.legend()
>>> plt.show()
```
inv(a)
Compute the (multiplicative) inverse of a matrix.

Given a square matrix a, return the matrix ainv satisfying dot(a, ainv) = dot(ainv, a) =
eye(a.shape[0]).

Parameters
a : array_like, shape (M, M)
Matrix to be inverted.

Returns
ainv : ndarray or matrix, shape (M, M)
(Multiplicative) inverse of the matrix a.

Raises
LinAlgError :
If a is singular or not square.

Examples
>>> from numpy import linalg as LA
>>> a = np.array([[1., 2.], [3., 4.]]
>>> ainv = LA.inv(a)
>>> np.allclose(np.dot(a, ainv), np.eye(2))
True
>>> np.allclose(np.dot(ainv, a), np.eye(2))
True

If a is a matrix object, then the return value is a matrix as well:

>>> ainv = LA.inv(np.matrix(a))
>>> ainv
matrix([[[-2. , 1. ],
          [1.5, -0.5]]])

pinv(a, rcond=1.0000000000000001e-15)
Compute the (Moore-Penrose) pseudo-inverse of a matrix.
Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all large singular values.

**Parameters**
- `a`: array_like, shape (M, N)
  Matrix to be pseudo-inverted.
- `rcond`: float
  Cutoff for small singular values. Singular values smaller (in modulus) than `rcond * largest_singular_value` (again, in modulus) are set to zero.

**Returns**
- `B`: ndarray, shape (N, M)
  The pseudo-inverse of `a`. If `a` is a matrix instance, then so is `B`.

**Raises**
- `LinAlgError`:
  If the SVD computation does not converge.

**Notes**
The pseudo-inverse of a matrix `A`, denoted `A^+`, is defined as: “the matrix that ‘solves’ [the least-squares problem] `Ax = b`, i.e., if `x` is said solution, then `A^+` is that matrix such that `x = A^+ b`.

It can be shown that if `Q_1 \Sigma Q_2^T = A` is the singular value decomposition of `A`, then `A^+ = Q_2 \Sigma^+ Q_1^T`, where `Q_{1,2}` are orthogonal matrices, `\Sigma` is a diagonal matrix consisting of `A`'s so-called singular values, (followed, typically, by zeros), and then `\Sigma^+` is simply the diagonal matrix consisting of the reciprocals of `A`'s singular values (again, followed by zeros). [R49]

**References**
[R49]

**Examples**
The following example checks that `a * a+ * a == a` and `a+ * a * a+ == a+`:

```python
>>> a = np.random.randn(9, 6)
>>> B = np.linalg.pinv(a)
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
True
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True
```

**tensorinv** *(a, ind=2)*
Compute the ‘inverse’ of an N-dimensional array.

The result is an inverse for `a` relative to the tensordot operation `tensordot(a, b, ind)`, i.e., up to floating-point accuracy, `tensordot(tensorinv(a), a, ind)` is the “identity” tensor for the tensordot operation.

**Parameters**
- `a`: array_like
  Tensor to ‘invert’. Its shape must be ‘square’, i.e., `prod(a.shape[:ind]) == prod(a.shape[ind:])`.
- `ind`: int, optional
Number of first indices that are involved in the inverse sum. Must be a positive integer, default is 2.

Returns

b : ndarray
   a’s tensordot inverse, shape a.shape[:ind] + a.shape[ind:].

Raises

LinAlgError :
   If a is singular or not ‘square’ (in the above sense).

See Also:
tensordot, tensorsolve

Examples

>>> a = np.eye(4, 6)
>>> a.shape = (4, 6, 8, 3)
>>> ainv = np.linalg.tensorinv(a, ind=2)
>>> ainv.shape
(8, 3, 4, 6)
>>> b = np.random.randn(4, 6)
>>> np.allclose(np.tensordot(ainv, b), np.linalg.tensorsolve(a, b))
True

>>> a = np.eye(4, 6)
>>> a.shape = (24, 8, 3)
>>> ainv = np.linalg.tensorinv(a, ind=1)
>>> ainv.shape
(8, 3, 24)
>>> b = np.random.randn(24)
>>> np.allclose(np.tensordot(ainv, b, 1), np.linalg.tensorsolve(a, b))
True

3.7.6 Exceptions

linalg.LinAlgError | Generic Python-exception-derived object raised by linalg functions.

exception LinAlgError
   Generic Python-exception-derived object raised by linalg functions.

   General purpose exception class, derived from Python’s exception.Exception class, programmatically raised in linalg functions when a Linear Algebra-related condition would prevent further correct execution of the function.

   Parameters

   None:

   Examples

>>> from numpy import linalg as LA
>>> LA.inv(np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
  File ".../linalg.py", line 350,
    in inv return wrap(solve(a, identity(a.shape[0], dtye=a.dtype)))
3.8 Random sampling (**numpy.random**)

### 3.8.1 Simple random data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rand(d0, d1, ..., dn)</code></td>
<td>Random values in a given shape. Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).</td>
</tr>
<tr>
<td><code>randn([d1, ..., dn])</code></td>
<td>Return a sample (or samples) from the “standard normal” distribution. If positive, int_like or int-convertible arguments are provided, <code>randn</code> generates an array of shape (d1, ..., dn), filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.</td>
</tr>
<tr>
<td><code>randint(low[, high, size])</code></td>
<td>Return random integers from low (inclusive) to high (exclusive).</td>
</tr>
<tr>
<td><code>random_integers(low[, high, size])</code></td>
<td>Return random integers between low and high, inclusive.</td>
</tr>
<tr>
<td><code>random_sample([size])</code></td>
<td>Return random floats in the half-open interval [0.0, 1.0).</td>
</tr>
<tr>
<td><code>bytes(length)</code></td>
<td>Return random bytes.</td>
</tr>
</tbody>
</table>

#### rand()  
Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

**Parameters**
- `d0, d1, ..., dn`: int
  - Shape of the output.

**Returns**
- `out`: ndarray, shape (d0, d1, ..., dn)
  - Random values.

**See Also:**
random

**Notes**

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to `random`.

**Examples**

```python
>>> np.random.rand(3,2)
array([[ 0.14022471, 0.96360618],
       [ 0.37601032, 0.25528411],
       [ 0.49313049, 0.94909878]])
```

**randn()**  
Return a sample (or samples) from the “standard normal” distribution.

If positive, int_like or int-convertible arguments are provided, `randn` generates an array of shape (d1, ..., dn), filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.
This is a convenience function. If you want an interface that takes a tuple as the first argument, use `numpy.random.standard_normal` instead.

**Parameters**

- `d1, ..., dn` : `n` ints, optional
  
The dimensions of the returned array, should be all positive.

**Returns**

- `Z` : ndarray or float
  
  A `(d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

**See Also:**

- `random.standard_normal`
  
  Similar, but takes a tuple as its argument.

**Notes**

For random samples from $N(\mu, \sigma^2)$, use:

\[
\sigma \times \text{np.random.randn}(...) + \mu
\]

**Examples**

```python
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from N(3, 6.25):

```python
>>> 2.5 * np.random.randn(2, 4) + 3
array([[ 0.39924804,  4.68456316,  4.99394529,  4.84057254],
       [ 4.00950034, -1.81814867,  7.29718677]]) #random
```

**randint**

Return random integers from `low` (inclusive) to `high` (exclusive).

Return random integers from the “discrete uniform” distribution in the “half-open” interval `[low, high)`. If `high` is `None` (the default), then results are from `[0, low)`. 

**Parameters**

- `low` : int
  
  Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the `highest` such integer).

- `high` : int, optional
  
  If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

- `size` : int or tuple of ints, optional
  
  Output shape. Default is None, in which case a single int is returned.

**Returns**

- `out` : int or ndarray of ints
  
  `size`-shaped array of random integers from the appropriate distribution, or a single such random int if `size` not provided.
See Also:

random.random_integers

similar to randint, only for the closed interval \([low, high]\), and 1 is the lowest value if \(high\) is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

Examples

```python
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])

>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```python
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

random_integers()

Return random integers between low and high, inclusive.

Return random integers from the “discrete uniform” distribution in the closed interval \([low, high]\). If \(high\) is None (the default), then results are from \([1, low]\).

Parameters

- **low**: int
  Lowest (signed) integer to be drawn from the distribution (unless \(high=None\), in which case this parameter is the highest such integer).

- **high**: int, optional
  If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if \(high=None\)).

- **size**: int or tuple of ints, optional
  Output shape. Default is None, in which case a single int is returned.

Returns

- **out**: int or ndarray of ints
  size-shaped array of random integers from the appropriate distribution, or a single such random int if size not provided.

See Also:

random.randint

Similar to random_integers, only for the half-open interval \([low, high]\), and 0 is the lowest value if \(high\) is omitted.

Notes

To sample from \(N\) evenly spaced floating-point numbers between \(a\) and \(b\), use:

\[
a + (b - a) \times (\text{np.random.random_integers}(N) - 1) / (N - 1.)
\]
Examples

```python
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3.,2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```python
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625, 1.25, 0.625, 0.625, 2.5])
```

Roll two six sided dice 1000 times and sum the results:

```python
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

**random_sample()**

Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of `random_sample` by $(b-a)$ and add $a$:

$$(b - a) \times \text{random_sample()} + a$$
Parameters

size : int or tuple of ints, optional

Defines the shape of the returned array of random floats. If None (the default), returns a single float.

Returns

out : float or ndarray of floats

Array of random floats of shape size (unless size=None, in which case a single float is returned).

Examples

>>> np.random.random_sample()
0.47108547995356098

>>> type(np.random.random_sample())
<type 'float'>

>>> np.random.random_sample((5,))
array([ 0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])

Three-by-two array of random numbers from [-5, 0):

>>> 5 * np.random.random_sample((3, 2)) - 5
array([[ 3.99149989, -0.52338984],
        [-2.99091858, -0.79479508],
        [-1.23204345, -1.75224494]])

bytes()

Return random bytes.

Parameters

length : int

Number of random bytes.

Returns

out : str

String of length N.

Examples

>>> np.random.bytes(10)
'eh\x85\x02SZ\xbf\xa4' #random

3.8.2 Permutations

shuffle(x)
Modify a sequence in-place by shuffling its contents.

permutation(x)
Randomly permute a sequence, or return a permuted range.

shuffle()
Modify a sequence in-place by shuffling its contents.

permutation()
Randomly permute a sequence, or return a permuted range.

Parameters

x : int or array_like
If \( x \) is an integer, randomly permute \( \text{np.arange}(x) \). If \( x \) is an array, make a copy and shuffle the elements randomly.

Returns

- **out**: ndarray

Permuted sequence or array range.

Examples

```python
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])

>>> np.random.permutation([1, 4, 9, 12, 15])
array([15, 1, 9, 4, 12])
```

### 3.8.3 Distributions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beta(a, b[, size])</code></td>
<td>The Beta distribution over ([0, 1]). Draw samples from a binomial distribution.</td>
</tr>
<tr>
<td><code>binomial(n, p[, size])</code></td>
<td>Draw samples from a binomial distribution.</td>
</tr>
<tr>
<td><code>chisquare(df[, size])</code></td>
<td>Draw samples from the Dirichlet distribution.</td>
</tr>
<tr>
<td><code>mtrand.dirichlet(alpha[, size])</code></td>
<td>Exponential distribution. Draw samples from a F distribution.</td>
</tr>
<tr>
<td><code>exponential([scale, size])</code></td>
<td>Draw samples from a F distribution. Draw samples from a Gamma distribution.</td>
</tr>
<tr>
<td><code>gamma(shape[, scale, size])</code></td>
<td>Draw samples from the geometric distribution.</td>
</tr>
<tr>
<td><code>geometric(p[, size])</code></td>
<td>Gumbel distribution.</td>
</tr>
<tr>
<td><code>gumbel([loc, scale, size])</code></td>
<td>Draw samples from a Hypergeometric distribution.</td>
</tr>
<tr>
<td><code>hypergeometric(ngood, nbad, nsample[, size])</code></td>
<td>Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).</td>
</tr>
<tr>
<td><code>laplace([loc, scale, size])</code></td>
<td>Draw samples from a Logistic distribution.</td>
</tr>
<tr>
<td><code>lognormal([mean, sigma, size])</code></td>
<td>Return samples drawn from a log-normal distribution.</td>
</tr>
<tr>
<td><code>logseries(p[, size])</code></td>
<td>Draw samples from a Logarithmic Series distribution.</td>
</tr>
<tr>
<td><code>multinomial(n, pvals[, size])</code></td>
<td>Draw samples from a multinomial distribution.</td>
</tr>
<tr>
<td><code>multivariate_normal(mean, cov[, size])</code></td>
<td>Draw random samples from a multivariate normal distribution.</td>
</tr>
<tr>
<td><code>negative_binomial(n, p[, size])</code></td>
<td>Draw samples from a negative_binomial distribution.</td>
</tr>
<tr>
<td><code>noncentral_chisquare(df, nonc[, size])</code></td>
<td>Draw samples from a noncentral chi-square distribution.</td>
</tr>
<tr>
<td><code>noncentral_f(dfnum, dfden, nonc[, size])</code></td>
<td>Draw samples from the noncentral F distribution.</td>
</tr>
<tr>
<td><code>normal([loc, scale, size])</code></td>
<td>Draw random samples from a normal (Gaussian) distribution.</td>
</tr>
<tr>
<td><code>pareto(a[, size])</code></td>
<td>Draw samples from a Pareto distribution with specified shape.</td>
</tr>
<tr>
<td><code>poisson([lam, size])</code></td>
<td>Draw samples from a Poisson distribution.</td>
</tr>
<tr>
<td><code>power(a[, size])</code></td>
<td>Draws samples in ([0, 1]) from a power distribution with positive exponent a - 1.</td>
</tr>
<tr>
<td><code>rayleigh([scale, size])</code></td>
<td>Draw samples from a Rayleigh distribution.</td>
</tr>
<tr>
<td><code>standard_cauchy([size])</code></td>
<td>Standard Cauchy distribution with mode = 0.</td>
</tr>
<tr>
<td><code>standard_exponential([size])</code></td>
<td>Draw samples from the standard exponential distribution.</td>
</tr>
<tr>
<td><code>standard_gamma([shape, size])</code></td>
<td>Draw samples from a Standard Gamma distribution.</td>
</tr>
<tr>
<td><code>standard_normal([size])</code></td>
<td>Returns samples from a Standard Normal distribution (mean=0, stdev=1).</td>
</tr>
<tr>
<td><code>standard_t(df[, size])</code></td>
<td>Standard Student’s t distribution with df degrees of freedom.</td>
</tr>
<tr>
<td><code>triangular(left, mode, right[, size])</code></td>
<td>Draw samples from the triangular distribution.</td>
</tr>
<tr>
<td><code>uniform([low, high, size])</code></td>
<td>Draw samples from a uniform distribution.</td>
</tr>
<tr>
<td><code>vonmises([mu, kappa, size])</code></td>
<td>Draw samples from a von Mises distribution.</td>
</tr>
<tr>
<td><code>wald(mean, scale[, size])</code></td>
<td>Draw samples from a Wald, or Inverse Gaussian, distribution.</td>
</tr>
<tr>
<td><code>weibull(a[, size])</code></td>
<td>Weibull distribution.</td>
</tr>
</tbody>
</table>
Table 3.1 – continued from previous page

| zipf(a[, size]) | Draw samples from a Zipf distribution. |

**beta()**

The Beta distribution over $[0, 1]$.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1} (1 - x)^{\beta-1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1} (1 - t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

**Parameters**

- **a**: float
  
  Alpha, non-negative.

- **b**: float
  
  Beta, non-negative.

- **size**: tuple of ints, optional
  
  The number of samples to draw. The output is packed according to the size given.

**Returns**

- **out**: ndarray
  
  Array of the given shape, containing values drawn from a Beta distribution.

**binomial()**

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer $> 0$ and p is in the interval $[0, 1]$. (n may be input as a float, but it is truncated to an integer in use)

**Parameters**

- **n**: float (but truncated to an integer)
  
  parameter, $> 0$.

- **p**: float
  
  parameter, $\geq 0$ and $\leq 1$.

- **size**: {tuple, int}
  
  Output shape. If the given shape is, e.g., $(m, n, k)$, then $m \times n \times k$ samples are drawn.

**Returns**

- **samples**: {ndarray, scalar}
where the values are all integers in $[0, n]$.

See Also:

scipy.stats.distributions.binom
probability density function, distribution or cumulative density function, etc.

Notes
The probability density for the Binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where $n$ is the number of trials, $p$ is the probability of success, and $N$ is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product $p * n \leq 5$, where $p$ = population proportion estimate, and $n$ = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then $p = 4/15 = 27\%$. $0.27 * 15 = 4$, so the binomial distribution should be used in this case.

References
[R69], [R70], [R71], [R72], [R73]

Examples
Draw samples from the distribution:

```python
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let’s do 20,000 trials of the model, and count the number that generate zero positive results.

```python
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

chisquare() Draw samples from a chi-square distribution.

```
When $df$ independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Parameters

df : int
Number of degrees of freedom.

size : tuple of ints, int, optional
Size of the returned array. By default, a scalar is returned.

Returns

output : ndarray
```

464 Chapter 3. Routines
Samples drawn from the distribution, packed in a size-shaped array.

**Raises**

ValueError:

When df <= 0 or when an inappropriate size (e.g. size=-1) is given.

**Notes**

The variable obtained by summing the squares of df independent, standard normally distributed random variables:

\[ Q = \sum_{i=0}^{df} X_i^2 \]

is chi-square distributed, denoted

\[ Q \sim \chi^2_k. \]

The probability density function of the chi-squared distribution is

\[ p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}, \]

where \( \Gamma \) is the gamma function,

\[ \Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt. \]

**References**

[R74], [R75]

**Examples**

```python
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

dirichlet()

Draw samples from the Dirichlet distribution.

Draw size samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial in Bayesian inference.

**Parameters**

- **alpha**: array
  Parameter of the distribution (k dimension for sample of dimension k).

- **size**: array
  Number of samples to draw.
Notes

\[ X \approx \prod_{i=1}^{k} x_i^{\alpha_i - 1} \]

Uses the following property for computation: for each dimension, draw a random sample \( y_i \) from a standard gamma generator of shape \( \alpha_i \), then \( X = \frac{1}{\sum_{i=1}^{k} y_i} (y_1, \ldots, y_n) \) is Dirichlet distributed.

References

[R103]

**exponential()**

Exponential distribution.

Its probability density function is

\[ f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right), \]

for \( x > 0 \) and 0 elsewhere. \( \beta \) is the scale parameter, which is the inverse of the rate parameter \( \lambda = 1/\beta \). The rate parameter is an alternative, widely used parameterization of the exponential distribution [R78].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [R76], or the time between page requests to Wikipedia [R77].

**Parameters**

- **scale**: float
  
  The scale parameter, \( \beta = 1/\lambda \).

- **size**: tuple of ints
  
  Number of samples to draw. The output is shaped according to \( size \).

References

[R76], [R77], [R78]

**f()**

Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, \( dfnum \) (degrees of freedom in numerator) and \( dfden \) (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

**Parameters**

- **dfnum**: float
  
  Degrees of freedom in numerator. Should be greater than zero.

- **dfden**: float
  
  Degrees of freedom in denominator. Should be greater than zero.

- **size**: {tuple, int}, optional
Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn. By default only one sample is returned.

Returns

samples : {ndarray, scalar}

Samples from the Fisher distribution.

See Also:

scipy.stats.distributions.f

probability density function, distribution or cumulative density function, etc.

Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable \(dfnum\) is the number of samples minus one, the between-groups degrees of freedom, while \(dfden\) is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

References

[R79], [R80]

Examples

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children’s blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```python
>>> dfnum = 1.  # between group degrees of freedom
>>> dfden = 48.  # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```python
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

\(\gamma\) (gamma)

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, \(shape\) (sometimes designated “k”) and \(scale\) (sometimes designated “theta”), where both parameters are > 0.

Parameters

- shape : scalar > 0
  
The shape of the gamma distribution.

- scale : scalar > 0, optional
  
The scale of the gamma distribution. Default is equal to 1.

- size : shape_tuple, optional
  

3.8. Random sampling (numpy.random)
Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

**Returns**

- **out**: ndarray, float

  Returns one sample unless `size` parameter is specified.

**See Also:**

- `scipy.stats.distributions.gamma`
  probability density function, distribution or cumulative density function, etc.

**Notes**

The probability density for the Gamma distribution is

\[
p(x) = x^{k-1} e^{-x/\theta} \frac{1}{\theta^k \Gamma(k)},
\]

where \(k\) is the shape and \(\theta\) the scale, and \(\Gamma\) is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

**References**

[R81], [R82]

**Examples**

Draw samples from the distribution:

```python
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) / (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```
geometric()

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, \( k = 1, 2, \ldots \).

The probability mass function of the geometric distribution is

\[
 f(k) = (1 - p)^{k-1} p
\]

where \( p \) is the probability of success of an individual trial.

**Parameters**

- **p**: float
  
The probability of success of an individual trial.

- **size**: tuple of ints
  
  Number of values to draw from the distribution. The output is shaped according to `size`.

**Returns**

- **out**: ndarray
  
  Samples from the geometric distribution, shaped according to `size`.

**Examples**

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```python
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```python
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```
gumbel()

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location (or mean) and scale (or standard deviation).

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with “exponential-like” tails, it may be derived by considering a Gaussian process of measurements, and generating the pdf for the maximum values from that set of measurements (see examples).

Parameters

- **loc**: float
  The location of the mode of the distribution.

- **scale**: float
  The scale parameter of the distribution.

- **size**: tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

See Also:

- scipy.stats.gumbel
  probability density function, distribution or cumulative density function, etc.

- weibull, scipy.stats.genextreme

Notes

The probability density for the Gumbel distribution is

\[
p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-(x-\mu)/\beta},
\]

where \(\mu\) is the mode, a location parameter, and \(\beta\) is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a “fat-tailed” distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of \(\mu + 0.57721/\beta\) and a variance of \(\frac{\pi^2}{6} \beta^2\).

References

[R83], [R84], [R85]

Examples

Draw samples from the distribution:
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...           * np.exp(-np.exp(-(bins - mu)/beta)),
...           linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```python
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...     a = np.random.normal(mu, beta, 1000)
...     means.append(a.mean())
...     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...           * np.exp(-np.exp(-(bins - mu)/beta)),
...           linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...           * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...           linewidth=2, color='g')
>>> plt.show()
```

**`hypergeometric()`**

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.
Parameters

- **ngood**: float (but truncated to an integer)
  parameter, > 0.

- **nbad**: float
  parameter, >= 0.

- **nsample**: float
  parameter, > 0 and <= ngood+nbad

- **size**: {tuple, int}
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns

- **samples**: {ndarray, scalar}
  where the values are all integers in [0, n].

See Also:

- scipy.stats.distributions.hypergeom
  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Hypergeometric distribution is

\[ P(x) = \binom{m}{x} \binom{N-m}{n-x} \binom{n}{N-n} \]

where 0 ≤ x ≤ m and n + m − N ≤ x ≤ n

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

References

[R86], [R87], [R88]

Examples

Draw samples from the distribution:

```python
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```
Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```python
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!
```

`laplace()`

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

**Parameters**

- `loc` : float
  - The position, \(\mu\), of the distribution peak.
- `scale` : float
  - \(\lambda\), the exponential decay.

**Notes**

It has the probability density function

\[
f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp \left( -\frac{|x - \mu|}{\lambda} \right).
\]

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

**References**

[R89], [R90], [R91], [R92]

**Examples**

Draw samples from the distribution

```python
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```python
>>> g = (1/(scale * np.sqrt(2 * np.pi))) * np.exp(- (x - loc)**2 / (2 * scale**2 ))
>>> plt.plot(x,g)
```
logistic()

Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Parameters

- **loc**: float
- **scale**: float > 0.
- **size**: {tuple, int}
  
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns

- **samples**: {ndarray, scalar}
  where the values are all integers in [0, n].

See Also:

`scipy.stats.distributions.logistic`

probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Logistic distribution is

\[
P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2},
\]

where \( \mu \) = location and \( s \) = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.
References

[R93], [R94], [R95]

Examples

Draw samples from the distribution:

```python
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
```

Draw samples from a log-normal distribution with specified mean, standard deviation, and shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

```python
lognormal()
```

Return samples drawn from a log-normal distribution.

```python
>>> def logist(x, loc, scale):
... return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
```

```python
lognormal()  # plot against distribution
```

Notes

A variable $x$ has a log-normal distribution if $\log(x)$ is normally distributed.

The probability density function for the log-normal distribution is

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{-\frac{(\ln(x) - \mu)^2}{2\sigma^2}}$$

where $\mu$ is the mean and $\sigma$ is the standard deviation of the normally distributed logarithm of the variable.

A log-normal distribution results if a random variable is the product of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the sum of a large number of independent, identically-distributed variables (see the last example). It is one of the so-called “fat-tailed” distributions.
The log-normal distribution is commonly used to model the lifespan of units with fatigue-stress failure modes. Since this includes most mechanical systems, the log-normal distribution has widespread application.

It is also commonly used to model oil field sizes, species abundance, and latent periods of infectious diseases.

**References**

[R96], [R97], [R98]

**Examples**

Draw samples from the distribution:

```python
>>> mu, sigma = 3., 1.  # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')

>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...       / (x * sigma * np.sqrt(2 * np.pi)))

>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```python
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.random(100)
...     b.append(np.product(a))

>>> b = np.array(b) / np.min(b)  # scale values to be positive

>>> count, bins, ignored = plt.hist(b, 100, normed=True, align='center')

>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))

>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...       / (x * sigma * np.sqrt(2 * np.pi)))

>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```
logseries()
Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0 < p < 1).

Parameters
- **loc** : float
- **scale** : float > 0.
- **size** : {tuple, int}
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns
- **samples** : {ndarray, scalar}
  where the values are all integers in [0, n].

See Also:
- scipy.stats.distributions.logser
  probability density function, distribution or cumulative density function, etc.

Notes
The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1 - p)}.$$

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References
- [R99], [R100], [R101], [R102]
Examples

Draw samples from the distribution:

```python
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution

```python
>>> def logseries(k, p):
...     return -p**k/(k*log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
logseries(bins, a).max(), 'r')
>>> plt.show()
```

multinomial()

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of \( p \) possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents \( n \) such experiments. Its values, \( X_i = [X_0, X_1, \ldots, X_p] \), represent the number of times the outcome was \( i \).

Parameters

- \( n \) : int
  - Number of experiments.
- \( pvals \) : sequence of floats, length \( p \)
  - Probabilities of each of the \( p \) different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as \( \text{sum}(pvals[:-1]) \leq 1 \)).
- \( size \) : tuple of ints
  - Given a \( size \) of \((M, N, K)\), then \( M*N*K \) samples are drawn, and the output shape becomes \((M, N, K, p)\), since each sample has shape \((p,)\).

Examples

Throw a dice 20 times:

```python
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```python
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:
```python
>>> np.random.multinomial(100, [1/7.]*5)
array([13, 16, 13, 16, 42])
```

**multivariate_normal()**

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalisation of the one-dimensional normal distribution to higher dimensions.

Such a distribution is specified by its mean and covariance matrix, which are analogous to the mean (average or “centre”) and variance (standard deviation squared or “width”) of the one-dimensional normal distribution.

**Parameters**

- **mean**: (N,) ndarray
  Mean of the N-dimensional distribution.

- **cov**: (N,N) ndarray
  Covariance matrix of the distribution.

- **size**: tuple of ints, optional
  Given a shape of, for example, (m,n,k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m,n,k,N). If no shape is specified, a single sample is returned.

**Returns**

- **out**: ndarray
  The drawn samples, arranged according to size. If the shape given is (m,n,...), then the shape of out is is (m,n,...,N).

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

**Notes**

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, \( X = [x_1, x_2, ... x_N] \). The covariance matrix element \( C_{ij} \) is the covariance of \( x_i \) and \( x_j \). The element \( C_{ii} \) is the variance of \( x_i \) (i.e. its “spread”).

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (\( \text{cov} \) is a multiple of the identity matrix)
- Diagonal covariance (\( \text{cov} \) has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```python
>>> mean = [0,0]
>>> cov = [[1,0],[0,1]]
```

```python
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
```

```python
>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.
References

[R104], [R105]

Examples

```python
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```python
>>> print list( (x[0,0,:] - mean) < 0.6 )
[True, True]
```

negative_binomial()

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, \( n \) trials and \( p \) probability of success where \( n \) is an integer > 0 and \( p \) is in the interval \([0, 1]\).

Parameters

- **n**: int
  - Parameter, > 0.
- **p**: float
  - Parameter, >= 0 and <=1.
- **size**: int or tuple of ints
  - Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

Returns

- **samples**: int or ndarray of ints
  - Drawn samples.

Notes

The probability density for the Negative Binomial distribution is

\[
P(N; n, p) = \binom{N + n - 1}{n - 1} p^n (1 - p)^N,\]

where \( n - 1 \) is the number of successes, \( p \) is the probability of success, and \( N + n - 1 \) is the number of trials.

The negative binomial distribution gives the probability of \( n-1 \) successes and \( N \) failures in \( N+n-1 \) trials, and success on the \((N+n)th\) trial.

If one throws a die repeatedly until the third time a “1” appears, then the probability distribution of the number of non-“1”s that appear before the third “1” is a negative binomial distribution.

References

[R106], [R107]
Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```python
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...    probability = sum(s<i) / 100000.
...    print i, "wells drilled, probability of one success =", probability
```

noncentral_chisquare()  

Draw samples from a noncentral chi-square distribution.

The noncentral \( \chi^2 \) distribution is a generalisation of the \( \chi^2 \) distribution.

Parameters

df : int

Degrees of freedom, should be \( \geq 1 \).

nonc : float

Non-centrality, should be \( > 0 \).

size : int or tuple of ints

Shape of the output.

Notes

The probability density function for the noncentral Chi-square distribution is

\[
P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-\frac{nonc}{2}}(\frac{nonc}{2})^i}{i!} \, Y_{df+2i}(x),
\]

where \( Y_q \) is the Chi-square with \( q \) degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

References

[R108], [R109]

Examples

Draw values from the distribution and plot the histogram

```python
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.
```python
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                     bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
noncentral_f()

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, dfnum (degrees of freedom in numerator) and dfden (degrees of freedom in denominator), where both parameters > 1. nonc is the non-centrality parameter.

Parameters
- dfnum : int
  Parameter, should be > 1.
- dfden : int
  Parameter, should be > 1.
- nonc : float
  Parameter, should be >= 0.
- size : int or tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns
- samples : scalar or ndarray
  Drawn samples.

Notes
When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

References
Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We’ll plot the two probability distributions for comparison.

```python
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

`normal()`

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [R111], is often called the bell curve because of its characteristic shape (see the example below).

The normal distribution occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [R111].

Parameters

- **loc**: float
  Mean (“centre”) of the distribution.

- **scale**: float
  Standard deviation (spread or “width”) of the distribution.

- **size**: tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

See Also:

- [scipy.stats.distributions.norm](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html)
  - probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

\[
p(x) = \frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}},
\]

where \(\mu\) is the mean and \(\sigma\) the standard deviation. The square of the standard deviation, \(\sigma^2\), is called the variance.

The function has its peak at the mean, and its “spread” increases with the standard deviation (the function reaches 0.607 times its maximum at \(x + \sigma\) and \(x - \sigma\) [R111]). This implies that `numpy.random.normal` is more likely to return samples lying close to the mean, rather than those far away.
**References**

[R110], [R111]

**Examples**

Draw samples from the distribution:

```python
>>> mu, sigma = 0, 0.1  # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```python
>>> abs(mu - np.mean(s)) < 0.01
True

>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...         np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...         linewidth=2, color='r')
>>> plt.show()
```

**pareto()**

Draw samples from a Pareto distribution with specified shape.

This is a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero. Most authors default the location to one.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the “80-20 rule”. In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.
Parameters

- **shape**: float, > 0.
  Shape of the distribution.

- **size**: tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

See Also:

- `scipy.stats.distributions.genpareto.pdf`
  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Pareto distribution is

\[ p(x) = \frac{am^a}{x^{a+1}} \]

where \( a \) is the shape and \( m \) the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called “fat-tailed” distributions.

References

[R112], [R113], [R114], [R115]

Examples

Draw samples from the distribution:

```python
g, m = 3., 1. # shape and mode
s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```python
import matplotlib.pyplot as plt
count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
fit = a*m**a/bins**(a+1)
plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
plt.show()
```
`poisson()`

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

**Parameters**

- `lam` : float
  Expectation of interval, should be >= 0.

- `size` : int or tuple of ints, optional
  Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

**Notes**

The Poisson distribution

\[
  f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}
\]

For events with an expected separation \( \lambda \) the Poisson distribution \( f(k; \lambda) \) describes the probability of \( k \) events occurring within the observed interval \( \lambda \).

**References**

[R116], [R117]

**Examples**

Draw samples from the distribution:

```python
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()

power()
Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
Also known as the power function distribution.

Parameters
a : float
    parameter, > 0
size : tuple of ints
    Output shape. If the given shape is, e.g., (m, n, k), then
    m * n * k samples are drawn.

Returns
samples : {ndarray, scalar}
    The returned samples lie in [0, 1].

Raises
ValueError:
    If a<1.

Notes
The probability density function is

\[ P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a \geq 0. \]

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.
It is used, for example, in modeling the over-reporting of insurance claims.
References

[R118], [R119]

Examples

Draw samples from the distribution:

```python
>>> a = 5.  # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```python
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```python
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```python
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```python
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```
inverse of 1 + np.random.pareto(5)

inverse of stats.pareto(5)

**rayleigh()**

Draw samples from a Rayleigh distribution.

The \( \chi \) and Weibull distributions are generalizations of the Rayleigh.

**Parameters**

- **scale**: scalar
  
  Scale, also equals the mode. Should be \( \geq 0 \).

- **size**: int or tuple of ints, optional
  
  Shape of the output. Default is None, in which case a single value is returned.
Notes

The probability density function for the Rayleigh distribution is

\[ P(x; \text{scale}) = \frac{x}{\text{scale}^\frac{3}{2}} e^{-\frac{x^2}{2\text{scale}^2}} \]

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

References


Examples

Draw values from the distribution and plot the histogram

```python
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```python
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```python
>>> 100.*sum(s>3)/1000000.
0.08730000000000003
```

standard_cauchy ()

Standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Parameters

- `size` : int or tuple of ints
  Shape of the output.

Returns

- `samples` : ndarray or scalar
  The drawn samples.

Notes

The probability density function for the full Cauchy distribution is

\[ P(x; x_0, \gamma) = \frac{1}{\pi \gamma [1 + \left( \frac{x - x_0}{\gamma} \right)^2]} \]
and the Standard Cauchy distribution just sets \( x_0 = 0 \) and \( \gamma = 1 \)

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

References


Examples

Draw samples and plot the distribution:

```python
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

`standard_exponential()`

Draw samples from the standard exponential distribution.

`standard_exponential` is identical to the exponential distribution with a scale parameter of 1.

Parameters

- `size`: int or tuple of ints
  Shape of the output.

Returns

- `out`: float or ndarray
  Drawn samples.

Examples

Output a 3x8000 array:

```python
>>> n = np.random.standard_exponential((3, 8000))
```

`standard_gamma()`

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated “k”) and scale=1.

Parameters

- `shape`: float
  Parameter, should be > 0.
- `size`: int or tuple of ints

3.8. Random sampling (`numpy.random`)
Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

**Returns**

- `samples`: ndarray or scalar
  The drawn samples.

**See Also:**

`scipy.stats.distributions.gamma`

probability density function, distribution or cumulative density function, etc.

**Notes**

The probability density for the Gamma distribution is

\[
p(x) = x^{k-1} e^{-x/\theta} \theta^k \Gamma(k),
\]

where \(k\) is the shape and \(\theta\) the scale, and \(\Gamma\) is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

**References**

[R121], [R122]

**Examples**

Draw samples from the distribution:

```python
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps

>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/
... (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```
**standard_normal()**

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

**Parameters**

- `size`: int or tuple of ints, optional
  
  Output shape. Default is None, in which case a single value is returned.

**Returns**

- `out`: float or ndarray
  
  Drawn samples.

**Examples**

```python
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893,  0.78096262, -0.89086505, ..., 0.49876311, #random
        -0.38672696, -0.4685006 ],) #random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

**standard_t()**

Standard Student’s t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As df gets large, the result resembles that of the standard normal distribution (`standard_normal`).

**Parameters**

- `df`: int
  
  Degrees of freedom, should be > 0.

- `size`: int or tuple of ints, optional
  
  Output shape. Default is None, in which case a single value is returned.
Returns

samples : ndarray or scalar

Drawn samples.

Notes

The probability density function for the t distribution is

\[
P(x, df) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df} \Gamma\left(\frac{df}{2}\right)} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}
\]

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to
test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gisset while working for the Guin-
ness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the
name Student.

References

[R123], [R124]

Examples

\texttt{triangular()}

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper
limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Parameters

left : scalar

Lower limit.

mode : scalar

The value where the peak of the distribution occurs. The value should fulfill the condi-
tion \texttt{left <= mode <= right}.

right : scalar

Upper limit, should be larger than \texttt{left}.

size : int or tuple of ints, optional

Output shape. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar

The returned samples all lie in the interval [left, right].

Notes

The probability density function for the Triangular distribution is

\[
P(x; l, m, r) = \begin{cases} 
\frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\
\frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\
0 & \text{otherwise}.
\end{cases}
\]
The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

References

http://en.wikipedia.org/wiki/Triangular_distribution

Examples

Draw values from the distribution and plot the histogram:

```python
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200, ...
...     normed=True)
>>> plt.show()
```

 uniform()  
Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval $[\text{low}, \text{high})$ (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by uniform.

Parameters

- low : float, optional
  Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

- high : float
  Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

- size : tuple of ints, int, optional
  Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

Returns

- out : ndarray
Drawn samples, with shape size.

See Also:

**randint**
Discrete uniform distribution, yielding integers.

**random_integers**
Discrete uniform distribution over the closed interval [low, high].

**random_sample**
Floats uniformly distributed over [0, 1).

**random**
Alias for `random_sample`.

**rand**
Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over [0, 1).

Notes
The probability density function of the uniform distribution is

\[ p(x) = \frac{1}{b - a} \]

anywhere within the interval \([a, b]\), and zero elsewhere.

Examples
Draw samples from the distribution:

```python
>>> s = np.random.uniform(-1, 0, 1000)
```

All values are within the given interval:

```python
>>> np.all(s >= -1)
True
```

```python
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```
vonmises()

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the
interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribu-
tion on the circle. It may be thought of as the circular analogue of the normal distribution.

Parameters

mu : float
   Mode (“center”) of the distribution.

kappa : float, >= 0.
   Dispersion of the distribution.

size : {tuple, int}
   Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are
drawn.

Returns

samples : {ndarray, scalar}
   The returned samples live on the unit circle [-pi, pi].

See Also:

scipy.stats.distributions.vonmises
probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the von Mises distribution is

\[ p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)} \],
where $\mu$ is the mode and $\kappa$ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises, named for Richard Edler von Mises, born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

References
[R125], [R126], [R127]

Examples
Draw samples from the distribution:

```python
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps

>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

wald()

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters

- mean : scalar
Distribution mean, should be > 0.

scale : scalar
   Scale parameter, should be >= 0.

size : int or tuple of ints, optional
   Output shape. Default is None, in which case a single value is returned.

Returns

   samples : ndarray or scalar
      Drawn sample, all greater than zero.

Notes

The probability density function for the Wald distribution is

\[ P(x; \text{mean}, \text{scale}) = \sqrt{\frac{\text{scale}}{2\pi x^3}} e^{-\frac{\text{scale}(x-\text{mean})^2}{2\text{mean}^2x}} \]

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

References


Examples

Draw values from the distribution and plot the histogram:

>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
weibull()
Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter.

\[ X = (-\ln(U))^{1/a} \]

Here, U is drawn from the uniform distribution over (0,1).

The more common 2-parameter Weibull, including a scale parameter \( \lambda \) is just \( X = \lambda(-\ln(U))^{1/a} \).

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

**Parameters**

- **a** : float
  Shape of the distribution.

- **size** : tuple of ints
  Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

**See Also:**

- scipy.stats.distributions.weibull
  probability density function, distribution or cumulative density function, etc.

- gumbel, scipy.stats.distributions.genextreme

**Notes**

The probability density for the Weibull distribution is

\[ p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a} \]
where $a$ is the shape and $\lambda$ the scale.

The function has its peak (the mode) at $\lambda(a-1)^{1/a}$.

When $a = 1$, the Weibull distribution reduces to the exponential distribution.

**References**

[R128], [R129], [R130]

**Examples**

Draw samples from the distribution:

```python
tax = 5.  # shape	s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1.,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)

>>> count, bins, ignored = plt.hist(np.random.weibull(5., 1000))
>>> x = np.arange(1.,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

**zipf()**

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter ($a$), where $a > 1$.

The zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf’s law, where the frequency of an item is inversely proportional to its rank in a frequency table.
Parameters

- **a**: float parameter, > 1.
- **size**: {tuple, int} Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns

- **samples**: {ndarray, scalar} The returned samples are greater than or equal to one.

See Also:

- `scipy.stats.distributions.zipf` probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Zipf distribution is

\[
p(x) = \frac{x^{-a}}{\zeta(a)},
\]

where \(\zeta\) is the Riemann Zeta function.

Named after the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

References

[R131], [R132], [R133], [R134]

Examples

Draw samples from the distribution:

```python
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> # Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```
3.8.4 Random generator

```python
mtrand.RandomState ([seed])  # Container for the Mersenne Twister pseudo-random number generator.
seed ([seed])                  # Seed the generator.
get_state ()                   # Return a tuple representing the internal state of the generator.
set_state (state)              # Set the internal state of the generator from a tuple.
```

class RandomState ()

Container for the Mersenne Twister pseudo-random number generator.

`RandomState` exposes a number of methods for generating random numbers drawn from a variety of probability distributions. In addition to the distribution-specific arguments, each method takes a keyword argument `size` that defaults to `None`. If `size` is `None`, then a single value is generated and returned. If `size` is an integer, then a 1-D array filled with generated values is returned. If `size` is a tuple, then an array with that shape is filled and returned.

**Parameters**

- **seed**: int or array_like, optional

  Random seed initializing the pseudo-random number generator. Can be an integer, an array (or other sequence) of integers of any length, or `None` (the default). If `seed` is `None`, then `RandomState` will try to read data from `/dev/urandom` (or the Windows analogue) if available or seed from the clock otherwise.

**Notes**

The Python stdlib module “random” also contains a Mersenne Twister pseudo-random number generator with a number of methods that are similar to the ones available in `RandomState`. `RandomState`, besides being NumPy-aware, has the advantage that it provides a much larger number of probability distributions to choose from.

**Methods**

- **seed ()**

  Seed the generator.
This method is called when `RandomState` is initialized. It can be called again to re-seed the generator. For details, see `RandomState`.

**Parameters**

- `seed`: int or array_like, optional  
  Seed for `RandomState`.

**See Also:**

- `RandomState`  
- `get_state()`  
  Return a tuple representing the internal state of the generator.  
  For more details, see `set_state`.  

**Returns**

- `out`: tuple(str, ndarray of 624 uints, int, int, float)  
  The returned tuple has the following items:  
  1. the string ‘MT19937’.
  2. a 1-D array of 624 unsigned integer keys.
  3. an integer `pos`.
  4. an integer `has_gauss`.
  5. a float `cached_gaussian`.

**See Also:**

- `set_state`

**Notes**

`set_state` and `get_state` are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

**set_state()**

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the “Mersenne Twister”[R120] pseudo-random number generating algorithm.

**Parameters**

- `state`: tuple(str, ndarray of 624 uints, int, int, float)  
  The `state` tuple has the following items:  
  1. the string ‘MT19937’, specifying the Mersenne Twister algorithm.
  2. a 1-D array of 624 unsigned integers `keys`.
  3. an integer `pos`.
  4. an integer `has_gauss`.
  5. a float `cached_gaussian`.

**Returns**

- `out`: None  
  Returns ‘None’ on success.
See Also:

get_state

Notes

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = (‘MT19937’, keys, pos).

References

[R120]

3.9 Sorting and searching

3.9.1 Sorting

```
| sort (a[, axis, kind, order])       | Return a sorted copy of an array. |
| lexxsort (keys[, axis])             | Perform an indirect sort using a sequence of keys. |
| argsort (a[, axis, kind, order])    | Returns the indices that would sort an array. |
| ndarray.sort ((axis, kind, order)) | Sort an array, in-place. |
| msort (a)                           | Return a copy of an array sorted along the first axis. |
| sort_complex (a)                    | Sort a complex array using the real part first, then the imaginary part. |
```

sort (a, axis=-1, kind=’quicksort’, order=None)

Return a sorted copy of an array.

Parameters

- **a**: array_like
  - Array to be sorted.

- **axis**: int or None, optional
  - Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

- **kind**: {‘quicksort’, ‘mergesort’, ‘heapsort’}, optional
  - Sorting algorithm. Default is ‘quicksort’.

- **order**: list, optional
  - When a is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

Returns

- **sorted_array**: ndarray
  - Array of the same type and shape as a.

See Also:

- ndarray.sort
  - Method to sort an array in-place.

- argsort
  - Indirect sort.
**lexsort**
Indirect stable sort on multiple keys.

**searchsorted**
Find elements in a sorted array.

**Notes**

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>~n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

Previous to numpy 1.4.0 sorting real and complex arrays containing nan values led to undefined behaviour. In numpy versions >= 1.4.0 nan values are sorted to the end. The extended sort order is:

- **Real**: [R, nan]
- **Complex**: [R + Rj, R + nanj, nan + Rj, nan + nanj]

where R is a non-nan real value. Complex values with the same nan placements are sorted according to the non-nan part if it exists. Non-nan values are sorted as before.

**Examples**

```python
>>> a = np.array([[1, 4], [3, 1]])
>>> np.sort(a)  # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)  # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)  # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the `order` keyword to specify a field to use when sorting a structured array:

```python
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ...
    ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)  # create a structured array
>>> np.sort(a, order='height')  # doctest: +SKIP
array([[('Galahad', 1.7, 38),
        ('Arthur', 1.8, 41),
        ('Lancelot', 1.9, 38)],
       dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

Sort by age, then height if ages are equal:
>> np.sort(a, order=[‘age’, ‘height’]) # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
       ('Arthur', 1.8, 41)],
dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')]

\textbf{lexsort()}

Perform an indirect sort using a sequence of keys.

Given multiple sorting keys, which can be interpreted as columns in a spreadsheet, \texttt{lexsort} returns an array of integer indices that describes the sort order by multiple columns. The last key in the sequence is used for the primary sort order, the second-to-last key for the secondary sort order, and so on. The keys argument must be a sequence of objects that can be converted to arrays of the same shape. If a 2D array is provided for the keys argument, its rows are interpreted as the sorting keys and sorting is according to the last row, second last row etc.

**Parameters**

\texttt{keys} : (k,N) array or tuple containing k (N,)-shaped sequences

The k different “columns” to be sorted. The last column (or row if \texttt{keys} is a 2D array) is the primary sort key.

\texttt{axis} : int, optional

Axis to be indirectly sorted. By default, sort over the last axis.

**Returns**

\texttt{indices} : (N,) ndarray of ints

Array of indices that sort the keys along the specified axis.

**See Also:**

\texttt{argsort}

Indirect sort.

\texttt{ndarray.sort}

In-place sort.

\texttt{sort}

Return a sorted copy of an array.

**Examples**

Sort names: first by surname, then by name.

```python
>>> surnames = ('Hertz', 'Galilei', 'Hertz')
>>> first_names = ('Heinrich', 'Galileo', 'Gustav')
>>> ind = np.lexsort((first_names, surnames))
>>> ind
array([1, 2, 0])
```

```python
>>> [surnames[i] + '
   , " + first_names[i] for i in ind]
['Galileo, Galileo', 'Hertz, Gustav', 'Hertz, Heinrich']
```

Sort two columns of numbers:
>>> a = [1,5,1,4,3,4] # First column
>>> b = [9,4,0,4,0,2] # Second column
>>> ind = np.lexsort((b,a)) # Sort by a, then by b
>>> print ind
[2 0 4 6 5 3 1]

>>> [(a[i],b[i]) for i in ind]
[(1, 0), (1, 9), (3, 0), (4, 1), (4, 2), (4, 4), (5, 4)]

Note that sorting is first according to the elements of a. Secondary sorting is according to the elements of b. A normal argsort would have yielded:

>>> [(a[i],b[i]) for i in np.argsort(a)]
[(1, 9), (1, 0), (3, 0), (4, 4), (4, 2), (4, 1), (5, 4)]

Structured arrays are sorted lexically by argsort:

>>> x = np.array([(1,9), (5,4), (1,0), (4,4), (3,0), (4,2), (4,1)],
...               dtype=np.dtype([('x', int), ('y', int)]))

>>> np.argsort(x) # or np.argsort(x, order=('x', 'y'))
array([2, 0, 4, 6, 5, 3, 1])

argsort (a, axis=-1, kind='quicksort', order=None)
Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as a that index data along the given axis in sorted order.

Parameters
a : array_like
  Array to sort.

axis : int or None, optional
  Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
  Sorting algorithm.

order : list, optional
  When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns
index_array : ndarray, int
  Array of indices that sort a along the specified axis. In other words, a[index_array] yields a sorted a.

See Also:
sort
  Describes sorting algorithms used.
**lexsort**
Indirect stable sort with multiple keys.

**ndarray.sort**
Inplace sort.

**Notes**
See *sort* for notes on the different sorting algorithms.

As of NumPy 1.4.0 *argsort* works with real/complex arrays containing nan values. The enhanced sort order is documented in *sort*.

**Examples**

One dimensional array:

```python
>>> x = np.array([3, 1, 2])
>>> np.argsort(x)
array([1, 2, 0])
```

Two-dimensional array:

```python
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])

>>> np.argsort(x, axis=0)
array([[0, 1],
       [1, 0]])

>>> np.argsort(x, axis=1)
array([[0, 1],
       [0, 1]])
```

Sorting with keys:

```python
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
      dtype=[('x', '<i4'), ('y', '<i4')])

>>> np.argsort(x, order=('x','y'))
array([1, 0])

>>> np.argsort(x, order=('y','x'))
array([0, 1])
```

**sort ()**
Sort an array, in-place.

**Parameters**
axis : int, optional
Axis along which to sort. Default is -1, which means sort along the last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
  Sorting algorithm. Default is 'quicksort'.

order : list, optional
  When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

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.

Notes

See sort for notes on the different sorting algorithms.

Examples

>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [3, 1]])

Use the order keyword to specify a field to use when sorting a structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])

msort(a)
  Return a copy of an array sorted along the first axis.

Parameters
  a : array_like
    Array to be sorted.

Returns
  sorted_array : ndarray
    Array of the same type and shape as a.
See Also:

sort

Notes

np.msort(a) is equivalent to np.sort(a, axis=0).

sort_complex(a)
Sort a complex array using the real part first, then the imaginary part.

Parameters

a : array_like
   Input array

Returns

out : complex ndarray
   Always returns a sorted complex array.

Examples

>>> np.sort_complex([5, 3, 6, 2, 1])
array([ 1.+0.j, 2.+0.j, 3.+0.j, 5.+0.j, 6.+0.j])

>>> np.sort_complex([1 + 2j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])
array([ 1.+2.j, 2.-1.j, 3.-5.j, 3.-3.j, 3.+2.j])

3.9.2 Searching

argmax(a[, axis])
Indices of the maximum values along an axis.

Parameters

a : array_like
   Input array.

axis : int, optional
   By default, the index is into the flattened array, otherwise along the specified axis.

Returns

index_array : ndarray of ints
   Array of indices into the array. It has the same shape as a.shape with the dimension along axis removed.

argmax(a, axis=None)
Indices of the maximum values along an axis.

Parameters

a : array_like
   Input array.

axis : int, optional
   By default, the index is into the flattened array, otherwise along the specified axis.

Returns

index_array : ndarray of ints
   Array of indices into the array. It has the same shape as a.shape with the dimension along axis removed.
See Also:

ndarray.argmax, argmin

amax
The maximum value along a given axis.

unravel_index
Convert a flat index into an index tuple.

Notes
In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])

>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b)  # Only the first occurrence is returned.
1
```

nanargmax (a, axis=None)
Return indices of the maximum values over an axis, ignoring NaNs.

Parameters

a : array_like
Input data.

axis : int, optional
Axis along which to operate. By default flattened input is used.

Returns

index_array : ndarray
An array of indices or a single index value.

See Also:
argmax, nanargmin

Examples

```python
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmax(a)
0
```
```python
>>> np.nanargmax(a)
1
>>> np.nanargmax(a, axis=0)
array([1, 0])
>>> np.nanargmax(a, axis=1)
array([1, 1])
```

**argmin** *(a, axis=None)*

Return the indices of the minimum values along an axis.

**See Also:**

**argmax**

Similar function. Please refer to `numpy.argmax` for detailed documentation.

**nanargmin** *(a, axis=None)*

Return indices of the minimum values over an axis, ignoring NaNs.

**Parameters**

- **a**: array_like
  
  Input data.

- **axis**: int, optional
  
  Axis along which to operate. By default flattened input is used.

**Returns**

- **index_array**: ndarray
  
  An array of indices or a single index value.

**See Also:**

argmin, nanargmax

**Examples**

```python
>>> a = np.array([[[np.nan, 4], [2, 3]]])
>>> np.argmin(a)
0
>>> np.nanargmin(a)
2
>>> np.nanargmin(a, axis=0)
array([1, 1])
>>> np.nanargmin(a, axis=1)
array([1, 0])
```

**argwhere** *(a)*

Find the indices of array elements that are non-zero, grouped by element.

**Parameters**

- **a**: array_like
  
  Input data.

**Returns**

- **index_array**: ndarray
  
  Indices of elements that are non-zero. Indices are grouped by element.
See Also:

where, nonzero

Notes

cpy.argwhere(a) is the same as np.transpose(np.nonzero(a)).
The output of argwhere is not suitable for indexing arrays. For this purpose use where(a) instead.

Examples

```python
>>> x = np.arange(6).reshape(2,3)
```
```
array([[0, 1, 2],
       [3, 4, 5]])
```
```
>>> np.argwhere(x>1)
```
```
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
```

nonzero(a)

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

Returns a tuple of arrays, one for each dimension of a, containing the indices of the non-zero elements in that
dimension. The corresponding non-zero values can be obtained with:

```
a[nonzero(a)]
```

To group the indices by element, rather than dimension, use:

```
transpose(nonzero(a))
```

The result of this is always a 2-D array, with a row for each non-zero element.

Parameters

a : array_like

    Input array.

Returns

tuple_of_arrays : tuple

    Indices of elements that are non-zero.

See Also:

flatnonzero

Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero

Equivalent ndarray method.

Examples

```python
>>> x = np.eye(3)
```
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```
```
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))

>>> x[np.nonzero(x)]
array([ 1., 1., 1.])

>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `np.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)

>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the boolean array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

```
flatnonzero(a)
Return indices that are non-zero in the flattened version of a.
This is equivalent to `a.ravel().nonzero()[0]`.

Parameters
a : ndarray
Input array.

Returns
res : ndarray
Output array, containing the indices of the elements of `a.ravel()` that are non-zero.

See Also:

nonzero
Return the indices of the non-zero elements of the input array.
ravel
Return a 1-D array containing the elements of the input array.
```

```python
flatnonzero(x)  # equivalent to x.ravel().nonzero()[0]
array([0, 1, 3, 4])
```
Use the indices of the non-zero elements as an index array to extract these elements:

```python
>>> x.ravel()[np.flatnonzero(x)]
array([-2, -1,  1,  2])
```

**where()**
Return elements, either from $x$ or $y$, depending on $condition$.

If only $condition$ is given, return $condition$.nonzero().

**Parameters**
- $condition$ : array_like, bool
  When True, yield $x$, otherwise yield $y$.
- $x$, $y$ : array_like, optional
  Values from which to choose. $x$ and $y$ need to have the same shape as $condition$.

**Returns**
- $out$ : ndarray or tuple of ndarrays
  If both $x$ and $y$ are specified, the output array contains elements of $x$ where $condition$ is True, and elements from $y$ elsewhere.
  If only $condition$ is given, return the tuple $condition$.nonzero(), the indices where $condition$ is True.

**See Also:**
- nonzero
- choose

**Notes**
If $x$ and $y$ are given and input arrays are 1-D, where is equivalent to:

```
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

**Examples**

```python
>>> np.where([[True, False], [True, True]],
          [[1, 2], [3, 4]],
          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])

>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))

>>> x = np.arange(9.).reshape(3, 3)
>>> np.where(x > 5)
(array([2, 2, 2]), array([0, 1, 2]))

>>> x[np.where(x > 3.0)]
array([4., 5., 6., 7., 8.])

>>> np.where(x < 5, x, -1)
array([[[0., 1., 2.],
       [3., 4., -1.],
       [-1., -1., -1.]]])
```

# Note: result is 1D.

# Note: broadcasting.
**searchsorted** *(a, v, side='left')*

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array *a* such that, if the corresponding elements in *v* were inserted before the indices, the order of *a* would be preserved.

**Parameters**

*a* : 1-D array_like
    Input array, sorted in ascending order.

*v* : array_like
    Values to insert into *a*.

*side* : {'left', 'right'}, optional
    If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or *N* (where *N* is the length of *a*).

**Returns**

*indices* : array of ints
    Array of insertion points with the same shape as *v*.

**See Also:**

- sort
  Return a sorted copy of an array.
- histogram
  Produce histogram from 1-D data.

**Notes**

Binary search is used to find the required insertion points.

As of Numpy 1.4.0 searchsorted works with real/complex arrays containing *nan* values. The enhanced sort order is documented in sort.

**Examples**

```python
>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])
```

**extract** *(condition, arr)*

Return the elements of an array that satisfy some condition.

This is equivalent to np.compress(ravel(condition), ravel(arr)). If *condition* is boolean np.extract is equivalent to arr[condition].

**Parameters**

*condition* : array_like
    An array whose nonzero or True entries indicate the elements of *arr* to extract.

*arr* : array_like

---

3.9. Sorting and searching
Input array of the same size as condition.

See Also:
take, put, putmask, compress

Examples

```python
>>> arr = np.arange(12).reshape((3, 4))
>>> arr
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 9, 10, 11]])
>>> condition = np.mod(arr, 3)==0
>>> condition
array([[ True, False, False,  True],
       [False, False,  True, False],
       [False,  True, False, False]], dtype=bool)
>>> np.extract(condition, arr)
array([0, 3, 6, 9])
If condition is boolean:

```python
>>> arr[condition]
array([0, 3, 6, 9])
```

3.10 Logic functions

3.10.1 Truth value testing

```python
all(a[, axis, out])  Test whether all array elements along a given axis evaluate to True.
any(a[, axis, out])  Test whether any array element along a given axis evaluates to True.
```
See Also:

**ndarray.all**

equivalent method

**any**

Test whether any array element along a given axis evaluates to True.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

```python
>>> np.all([[True, False], [True, True]])
False

>>> np.all([[True, False], [True, True]], axis=0)
array([ True, False], dtype=bool)

>>> np.all([-1, 4, 5])
True

>>> np.all([1.0, np.nan])
True

>>> o=np.array([False])
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array([ True], dtype=bool))
```

**any (a, axis=None, out=None)**

Test whether any array element along a given axis evaluates to True.

Returns single boolean unless `axis` is not `None`

Parameters

- **a**: array_like
  
  Input array or object that can be converted to an array.

- **axis**: int, optional
  
  Axis along which a logical OR is performed. The default (axis = None) is to perform a logical OR over a flattened input array. `axis` may be negative, in which case it counts from the last to the first axis.

- **out**: ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved. See doc.ufuncs (Section “Output arguments”) for details.

Returns

- **any**: bool, ndarray
  
  A new boolean or `ndarray` is returned unless `out` is specified, in which case a reference to `out` is returned.
See Also:

ndarray.any
equivalent method

all
Test whether all array elements along a given axis evaluate to True.

Notes
Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

```python
>>> np.any([[True, False], [True, True]])
True

>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False], dtype=bool)

>>> np.any([-1, 0, 5])
True

>>> np.any(np.nan)
True

>>> o=np.array([False])
>>> z=np.any([-1, 0, 5], out=o)
>>> z, o
(array([ True], dtype=bool), array([ True], dtype=bool))

>>> # Check now that z is a reference to o
>>> z
is o
True

>>> id(z), id(o) # identity of z and o
(191614240, 191614240)
```

3.10.2 Array contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isfinite(x[, out])</code></td>
<td>Test element-wise for finite-ness (not infinity or not Not a Number).</td>
</tr>
<tr>
<td><code>isinf(x[, out])</code></td>
<td>Test element-wise for positive or negative infinity, return result as bool array.</td>
</tr>
<tr>
<td><code>isnan(x[, out])</code></td>
<td>Test element-wise for Not a Number (NaN), return result as a bool array.</td>
</tr>
<tr>
<td><code>isneginf(x, y)</code></td>
<td>Test element-wise for negative infinity, return result as bool array.</td>
</tr>
<tr>
<td><code>isposinf(x, y)</code></td>
<td>Test element-wise for positive infinity, return result as bool array.</td>
</tr>
</tbody>
</table>

**isfinite()**
Test element-wise for finite-ness (not infinity or not Not a Number).

The result is returned as a boolean array.

**Parameters**

- **x**: array_like
  Input values.

- **out**: ndarray, optional
Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

y : ndarray, bool

For scalar input, the result is a new boolean with value True if the input is finite; otherwise the value is False (input is either positive infinity, negative infinity or Not a Number).

For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is finite; otherwise the values are False (element is either positive infinity, negative infinity or Not a Number).

See Also:

isinf, isneginf, isposinf, isnan

Notes

Not a Number, positive infinity and negative infinity are considered to be non-finite.

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity. Errors result if the second argument is also supplied when x is a scalar input, or if first and second arguments have different shapes.

Examples

```python
gnp.isfinite(1)
True
gnp.isfinite(0)
True
gnp.isfinite(np.nan)
False
gnp.isfinite(np.inf)
False
gnp.isfinite(np.NINF)
False
gnp.isfinite([np.log(-1.), 1., np.log(0)])
array([False,  True, False], dtype=bool)
```

```python
x = np.array([-np.inf, 0., np.inf])
y = np.array([2, 2, 2])
gnp.isfinite(x, y)
array([0, 1, 0])
y
array([0, 1, 0])
```

isinf()

Test element-wise for positive or negative infinity, return result as bool array.

Parameters

x : array_like

Input values

y : array_like, optional

An array with the same shape as x to store the result.
Returns

\[ y : \{ \text{ndarray, bool} \} \]

For scalar input, the result is a new boolean with value True if the input is positive or negative infinity; otherwise the value is False.

For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is positive or negative infinity; otherwise the values are False. If a second argument is supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True. The return value \( y \) is then a reference to that array.

See Also:

\texttt{isnan}, \texttt{isposinf}, \texttt{isneginf}, \texttt{isfinite}

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if second argument is also supplied with scalar input or if first and second arguments have different shapes.

Examples

```python
>>> np.isinf(np.inf)
True
>>> np.isinf(np.nan)
False
>>> np.isinf(np.NINF)
True
>>> np.isinf([np.inf, -np.inf, 1.0, np.nan])
array([ True,  True, False, False], dtype=bool)
```

```python
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isinf(x, y)
array([1, 0, 1])
>>> y
array([1, 0, 1])
```

\texttt{isnan()}

Test element-wise for Not a Number (NaN), return result as a bool array.

Parameters

\[ x : \text{array_like} \]

Input array.

Returns

\[ y : \{ \text{ndarray, bool} \} \]

For scalar input, the result is a new boolean with value True if the input is NaN; otherwise the value is False.

For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is NaN; otherwise the values are False.
See Also:

`isinf`, `isneginf`, `isposinf`, `isfinite`

Notes
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```python
>>> np.isnan(np.nan)
True
>>> np.isnan(np.inf)
False
>>> np.isnan([np.log(-1.),1.,np.log(0)])
array([ True, False, False], dtype=bool)
```

`isneginf`(x, y=None)
Test element-wise for negative infinity, return result as bool array.

Parameters
- `x`: array_like
  The input array.
- `y`: array_like, optional
  A boolean array with the same shape and type as `x` to store the result.

Returns
- `y`: ndarray
  A boolean array with the same dimensions as the input. If second argument is not supplied then a numpy boolean array is returned with values True where the corresponding element of the input is negative infinity and values False where the element of the input is not negative infinity.

Errors result if the second argument is also supplied when x is a scalar input, or if first and second arguments have different shapes.

```python
>>> np.isneginf(np.NINF)
array(True, dtype=bool)
>>> np.isneginf(np.inf)
array(False, dtype=bool)
>>> np.isneginf(np.PINF)
array(False, dtype=bool)
```
>>> np.isneginf([-np.inf, 0., np.inf])
array([True, False, False], dtype=bool)

>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isneginf(x, y)
array([1, 0, 0])

isneginf(x, y=None)
Test element-wise for negative infinity, return result as bool array.

Parameters
x : array_like
The input array.
y : array_like, optional
A boolean array with the same shape as x to store the result.

Returns
y : ndarray
A boolean array with the same dimensions as the input. If second argument is not supplied then a boolean array is returned with values True where the corresponding element of the input is negative infinity and values False where the element of the input is not negative infinity.

If a second argument is supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True. The return value y is then a reference to that array.

See Also:
isinf, isneginf, isfinite, isnan

Notes
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).
Errors result if the second argument is also supplied when x is a scalar input, or if first and second arguments have different shapes.

Examples

>>> np.isposinf(np.PINF)
array(True, dtype=bool)
>>> np.isposinf(np.inf)
array(True, dtype=bool)
>>> np.isposinf(np.NINF)
array(False, dtype=bool)
>>> np.isposinf([-np.inf, 0., np.inf])
array([False, False, True], dtype=bool)

>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isposinf(x, y)
3.10.3 Array type testing

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iscomplex(x)</code></td>
<td>Returns a bool array, where True if input element is complex.</td>
</tr>
<tr>
<td><code>iscomplexobj(x)</code></td>
<td>Return True if x is a complex type or an array of complex numbers.</td>
</tr>
<tr>
<td><code>isfortran(a)</code></td>
<td>Returns True if array is arranged in Fortran-order in memory and dimension &gt; 1.</td>
</tr>
<tr>
<td><code>isreal(x)</code></td>
<td>Returns a bool array, where True if input element is real.</td>
</tr>
<tr>
<td><code>isrealobj(x)</code></td>
<td>Return True if x is a not complex type or an array of complex numbers.</td>
</tr>
<tr>
<td><code>isscalar(num)</code></td>
<td>Returns True if the type of num is a scalar type.</td>
</tr>
</tbody>
</table>

`iscomplex(x)`

Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

**Parameters**
- `x`: array_like
  - Input array.

**Returns**
- `out`: ndarray of bools
  - Output array.

**See Also:**
- `isreal`
- `iscomplexobj`

`iscomplexobj(x)`

Return True if x is a complex type or an array of complex numbers.

The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, `iscomplexobj` evaluates to True if the data type is complex.

**Parameters**
- `x`: any
  - The input can be of any type and shape.

**Returns**
- `y`: bool
  - The return value, True if x is of a complex type.

**See Also:**
- `isrealobj, iscomplex`
Examples

>>> np.iscomplexobj(1)
False
>>> np.iscomplexobj(1+0j)
True
np.iscomplexobj([3, 1+0j, True])
True

isfortran(a)

Returns True if array is arranged in Fortran-order in memory and dimension > 1.

Parameters

a : ndarray
Input array.

Examples

np.array allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(a)
False

>>> b = np.array([[1, 2, 3], [4, 5, 6]], order='FORTRAN')
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(b)
True

The transpose of a C-ordered array is a FORTRAN-ordered array.

>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.isfortran(b)
True

1-D arrays always evaluate as False.

>>> np.isfortran(np.array([1, 2], order='FORTRAN'))
False
isreal\( (x) \)
Returns a bool array, where True if input element is real.
If element has complex type with zero complex part, the return value for that element is True.

Parameters
\( x \) : array_like
Input array.

Returns
\( out \) : ndarray, bool
Boolean array of same shape as \( x \).

See Also:
iscomplex

isrealobj
Return True if \( x \) is not a complex type.

Examples

```python
>>> np.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([False, True, True, True, True, False], dtype=bool)
```

isrealobj\( (x) \)
Return True if \( x \) is a not complex type or an array of complex numbers.
The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, isrealobj evaluates to False if the data type is complex.

Parameters
\( x \) : any
The input can be of any type and shape.

Returns
\( y \) : bool
The return value, False if \( x \) is of a complex type.

See Also:
iscomplexobj, isreal

Examples

```python
>>> np.isrealobj(1)
True
>>> np.isrealobj(1+0j)
False
>>> np.isrealobj([3, 1+0j, True])
False
```

isscalar\( (num) \)
Returns True if the type of \( num \) is a scalar type.

Parameters
\( num \) : any
Input argument, can be of any type and shape.
Returns
val : bool

True if `num` is a scalar type. False if it is not.

Examples

```python
>>> np.isscalar(3.1)
True
>>> np.isscalar([3.1])
False
>>> np.isscalar(False)
True
```

3.10.4 Logical operations

| logical_and | Computes the truth value of x1 AND x2 elementwise. |
| logical_or | Computes the truth value of x1 OR x2 elementwise. |
| logical_not | Computes the truth value of NOT x elementwise. |
| logical_xor | Computes the truth value of x1 XOR x2 elementwise. |

**logical_and**

Compute the truth value of x1 AND x2 elementwise.

Parameters

- `x1, x2` : array_like

  Input arrays. `x1` and `x2` must be of the same shape.

Returns

- `y` : ndarray, bool

  Boolean result with the same shape as `x1` and `x2` of the logical AND operation on corresponding elements of `x1` and `x2`.

See Also:

- `logical_or`, `logical_not`, `logical_xor`, `bitwise_and`

Examples

```python
>>> np.logical_and(True, False)
False
>>> np.logical_and([True, False], [False, False])
array([False, False], dtype=bool)

>>> x = np.arange(5)
>>> np.logical_and(x>1, x<4)
array([False, False, True, True, False], dtype=bool)
```

**logical_or**

Compute the truth value of x1 OR x2 elementwise.

Parameters

- `x1, x2` : array_like

  Logical OR is applied to the elements of `x1` and `x2`. They have to be of the same shape.
**Returns**

\[ y: \{\text{ndarray, bool}\} \]

Boolean result with the same shape as \( x1 \) and \( x2 \) of the logical OR operation on elements of \( x1 \) and \( x2 \).

**See Also:**

`logical_and`, `logical_not`, `logical_xor`, `bitwise_or`

**Examples**

```python
gap> np.logical_or(True, False)
True
gap> np.logical_or([True, False], [False, False])
array([ True, False], dtype=bool)
```

**logical_not**

Compute the truth value of NOT \( x \) elementwise.

**Parameters**

\[ x: \text{array_like} \]

Logical NOT is applied to the elements of \( x \).

**Returns**

\[ y: \{\text{ndarray, bool}\} \]

Boolean result with the same shape as \( x \) of the NOT operation on elements of \( x \).

**See Also:**

`logical_and`, `logical_or`, `logical_xor`

**Examples**

```python
gap> np.logical_not(3)
False
gap> np.logical_not([True, False, 0, 1])
array([False, True, True, False], dtype=bool)
```

```python
gap> x = np.arange(5)
gap> np.logical_not(x<3)
array([False, False, False, True, True], dtype=bool)
```

**logical_xor**

Compute the truth value of \( x1 \) XOR \( x2 \) elementwise.

**Parameters**

\[ x1, x2: \text{array_like} \]

Logical XOR is applied to the elements of \( x1 \) and \( x2 \). They have to be of the same shape.

**Returns**

\[ y: \{\text{ndarray, bool}\} \]
Boolean result with the same shape as \(x1\) and \(x2\) of the logical XOR operation on elements of \(x1\) and \(x2\).

See Also:
- \texttt{logical\_and}, \texttt{logical\_or}, \texttt{logical\_not}, \texttt{bitwise\_xor}

Examples

```python
>>> np.logical_xor(True, False)
True
>>> np.logical_xor([True, True, False, False], [True, False, True, False])
array([False, True, True, False], dtype=bool)
```

```python
>>> x = np.arange(5)
>>> np.logical_xor(x < 1, x > 3)
array([True, False, False, False, True], dtype=bool)
```

3.10.5 Comparison

<table>
<thead>
<tr>
<th>allclose ((a, b), rtol, atol)</th>
<th>Returns True if two arrays are element-wise equal within a tolerance.</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_equal ((a1, a2))</td>
<td>True if two arrays have the same shape and elements, False otherwise.</td>
</tr>
<tr>
<td>array_equiv ((a1, a2))</td>
<td>Returns True if input arrays are shape consistent and all elements equal.</td>
</tr>
</tbody>
</table>

**allclose** \((a, b, \text{rtol}=1.0000000000000001e-05, \text{atol}=1e-08)\)

Returns True if two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference \((\text{rtol} \times \text{abs}(b))\) and the absolute difference \(\text{atol}\) are added together to compare against the absolute difference between \(a\) and \(b\).

**Parameters**
- \(a, b\) : array_like
  Input arrays to compare.
- \texttt{rtol} : float
  The relative tolerance parameter (see Notes).
- \texttt{atol} : float
  The absolute tolerance parameter (see Notes).

**Returns**
- \(y\) : bool
  Returns True if the two arrays are equal within the given tolerance; False otherwise. If either array contains NaN, then False is returned.

See Also:
- \texttt{all}, \texttt{any}, \texttt{alltrue}, \texttt{sometrue}

**Notes**

If the following equation is element-wise True, then allclose returns True.

\[
\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} \times \text{absolute}(b))
\]

The above equation is not symmetric in \(a\) and \(b\), so that \texttt{allclose}(\(a, b\)) might be different from \texttt{allclose}(\(b, a\)) in some rare cases.
Examples

```python
>>> np.allclose([1e10,1e-7], [1.00001e10,1e-8])
False
>>> np.allclose([1e10,1e-8], [1.00001e10,1e-9])
True
>>> np.allclose([1e10,1e-8], [1.0001e10,1e-9])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan])
False
```

array_equal (a1, a2)

True if two arrays have the same shape and elements, False otherwise.

Parameters

- **a1, a2**: array_like
  
  Input arrays.

Returns

- **b**: bool
  
  Returns True if the arrays are equal.

See Also:

- **allclose**
  
  Returns True if two arrays are element-wise equal within a tolerance.

- **array_equiv**
  
  Returns True if input arrays are shape consistent and all elements equal.

Examples

```python
>>> np.array_equal([1, 2], [1, 2])
True
>>> np.array_equal(np.array([1, 2]), np.array([1, 2]))
True
>>> np.array_equal([1, 2], [1, 2, 3])
False
>>> np.array_equal([1, 2], [1, 4])
False
```

array_equiv (a1, a2)

Returns True if input arrays are shape consistent and all elements equal.

Shape consistent means they are either the same shape, or one input array can be broadcasted to create the same shape as the other one.

Parameters

- **a1, a2**: array_like
  
  Input arrays.

Returns

- **out**: bool
  
  True if equivalent, False otherwise.
Examples

```python
greater(x1, x2[, out])
greater_equal(x1, x2[, out])
less(x1, x2[, out])
less_equal(x1, x2[, out])
equal(x1, x2[, out])
not_equal(x1, x2[, out])
```

greater()  
Return (x1 > x2) element-wise.

Parameters
- x1, x2 : array_like

Returns
- Out : {ndarray, bool}

Examples

```python
>>> np.greater([4,2],[2,2])
array([ True, False], dtype=bool)
```

If the inputs are ndarrays, then np.greater is equivalent to '>'.

```python
>>> a = np.array([4,2])
>>> b = np.array([2,2])
>>> a > b
array([ True, False], dtype=bool)
```

greater_equal()  
Return (x1 >= x2) element-wise.

Parameters
- x1, x2 : array_like

Returns
- Out : {ndarray, bool}

Examples

```python
>>> np.greater_equal([4,2],[2,2])
array([ True, True], dtype=bool)
```
Returns

out : {ndarray, bool}

Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
greater, less, less_equal, equal, not_equal

Examples

>>> np.greater_equal([4, 2, 1], [2, 2, 2])
array([ True, True, False], dtype=bool)

less()
Return (x1 < x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays.

Returns

Out : ndarray of bools

Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
less_equal, greater, greater_equal, equal, not_equal

Examples

>>> np.less([1, 2], [2, 2])
array([ True, False], dtype=bool)

less_equal()
Return (x1 <= x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays.

Returns

Out : {ndarray, bool}

Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
less, greater_equal, greater, equal, not_equal

Examples

>>> np.less_equal([1, 2, 3], [2, 2, 2])
array([ True, True, False], dtype=bool)

equal()
Return (x1 == x2) element-wise.

Parameters

x1, x2 : array_like
Input arrays of the same shape.

Returns

**out**: {ndarray, bool}

Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:

*not_equal*, *greater_equal*, *less_equal*, *greater*, *less*

Examples

```python
>>> np.equal([0, 1, 3], np.arange(3))
array([ True,  True, False], dtype=bool)
```

What is compared are values, not types. So an int (1) and an array of length one can evaluate as True:

```python
>>> np.equal(1, np.ones(1))
array([ True], dtype=bool)
```

**not_equal ()**

Return (x1 != x2) element-wise.

Parameters

x1, x2 : array_like

Input arrays.

out : ndarray, optional

A placeholder the same shape as x1 to store the result. See doc.ufuncs (Section “Output arguments”) for more details.

Returns

**not_equal**: ndarray bool, scalar bool

For each element in x1, x2, return True if x1 is not equal to x2 and False otherwise.

See Also:

equal, greater, greater_equal, less, less_equal

Examples

```python
>>> np.not_equal([1., 2.], [1., 3.])
array([False, True], dtype=bool)
```

```python
>>> np.not_equal([1, 2], [[1, 3],[1, 4]])
array([[False, True],
       [False, True]], dtype=bool)
```
3.11 Binary operations

3.11.1 Elementwise bit operations

**bitwise_and**

Inputs:
- `x1, x2`: array_like

Outputs:
- `out`: array_like

**bitwise_or**

Inputs:
- `x1, x2`: array_like

Outputs:
- `out`: array_like

**bitwise_xor**

Inputs:
- `x1, x2`: array_like

Outputs:
- `out`: array_like

**invert**

Inputs:
- `x`: array_like

Outputs:
- `out`: array_like

**left_shift**

Inputs:
- `x1, x2`: array_like

Outputs:
- `out`: array_like

**right_shift**

Inputs:
- `x1, x2`: array_like

Outputs:
- `out`: array_like

### bitwise_and() Function

**Description**

Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `&`.

**Parameters**

- `x1, x2`: array_like

  Only integer types are handled (including booleans).

**Returns**

- `out`: array_like

  Result.

**See Also**

- `logical_and`, `bitwise_or`, `bitwise_xor`

**binary_repr**

Return the binary representation of the input number as a string.

**Examples**

The number 13 is represented by 00001101. Likewise, 17 is represented by 00010001. The bit-wise AND of 13 and 17 is therefore 00000001, or 1:

```python
>>> np.bitwise_and(13, 17)
1

>>> np.bitwise_and(14, 13)
12

>>> np.bitwise_and([14,3], 13)
array([12, 1])

>>> np.bitwise_and([11,7], [4,25])
array([0, 1])

>>> np.bitwise_and(np.array([2,5,255]), np.array([3,14,16]))
array([2, 4, 16])

>>> np.bitwise_and([True, True], [False, True])
array([False, True], dtype=bool)
```
**bitwise_or()**

Compute the bit-wise OR of two arrays element-wise.

Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `|`.

**Parameters**

- `x1, x2` : `array_like`
  
  Only integer types are handled (including booleans).

- `out` : `ndarray`, optional
  
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

**Returns**

- `out` : `array_like`
  
  Result.

**See Also:**

- `logical_or`, `bitwise_and`, `bitwise_xor`

**binary_repr**

Return the binary representation of the input number as a string.

**Examples**

The number 13 has the binary representation `00001101`. Likewise, 16 is represented by `00010000`. The bit-wise OR of 13 and 16 is then `00011101`, or 29:

```python
>>> np.bitwise_or(13, 16)
29
>>> np.binary_repr(29)
'11101'
```

```python
>>> np.bitwise_or(32, 2)
34
>>> np.bitwise_or([33, 4], 1)
array([33, 5])
>>> np.bitwise_or([33, 4], [1, 2])
array([33, 6])
```

```python
>>> np.bitwise_or(np.array([2, 5, 255]), np.array([4, 4, 4]))
array([ 6, 5, 255])
>>> np.array([2, 5, 255]) | np.array([4, 4, 4])
array([ 6, 5, 255])
>>> np.bitwise_or(np.array([2, 5, 255, 2147483647L], dtype=np.int32),
... np.array([4, 4, 4, 2147483647L], dtype=np.int32))
array([ 6, 5, 255, 2147483647])
```

```python
>>> np.bitwise_or([True, True], [False, True])
array([ True, True], dtype=bool)
```

**bitwise_xor()**

Compute the bit-wise XOR of two arrays element-wise.

Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `^`.
Parameters
   x1, x2 : array_like
       Only integer types are handled (including booleans).

Returns
   out : array_like
       Result.

See Also:
   logical_xor, bitwise_and, bitwise_or

binary_repr
   Return the binary representation of the input number as a string.

Examples
The number 13 is represented by 00001101. Likewise, 17 is represented by 00010001. The bitwise XOR of
13 and 17 is therefore 00011100, or 28:

>>> np.bitwise_xor(13, 17)
28
>>> np.binary_repr(28)
'11100'

>>> np.bitwise_xor(31, 5)
26
>>> np.bitwise_xor([31,3], 5)
array([26, 6])

>>> np.bitwise_xor([31,3], [5,6])
array([26, 5])
>>> np.bitwise_xor([True, True], [False, True])
array([ True, False], dtype=bool)

invert()
   Compute bit-wise inversion, or bit-wise NOT, element-wise.

   Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This
ufunc implements the C/Python operator ~.

   For signed integer inputs, the two’s complement is returned. In a two’s-complement system negative numbers
are represented by the two’s complement of the absolute value. This is the most common method of representing
signed integers on computers [R41]. A N-bit two’s-complement system can represent every integer in the range
$-2^{N-1}$ to $+2^{N-1} - 1$.

Parameters
   x1 : array_like
       Only integer types are handled (including booleans).

Returns
   out : array_like
       Result.

See Also:
   bitwise_and, bitwise_or, bitwise_xor, logical_not
**binary_repr**

Return the binary representation of the input number as a string.

**Notes**

`bitwise_not` is an alias for `invert`:

```python
>>> np.bitwise_not is np.invert
True
```

**References**

[R41]

**Examples**

We’ve seen that 13 is represented by 00001101. The invert or bit-wise NOT of 13 is then:

```python
>>> np.invert(np.array([13], dtype=uint8))
array([242], dtype=uint8)
>>> np.binary_repr(x, width=8)
'00001101'
>>> np.binary_repr(242, width=8)
'11110010'
```

The result depends on the bit-width:

```python
>>> np.invert(np.array([13], dtype=uint16))
array([65522], dtype=uint16)
>>> np.binary_repr(x, width=16)
'0000000000001101'
>>> np.binary_repr(65522, width=16)
'1111111111110010'
```

When using signed integer types the result is the two’s complement of the result for the unsigned type:

```python
>>> np.invert(np.array([13], dtype=int8))
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8)
'11110010'
```

Booleans are accepted as well:

```python
>>> np.invert(array([True, False]))
array([False, True], dtype=bool)
```

**left_shift()**

Shift the bits of an integer to the left.

Bits are shifted to the left by appending \(x_2\) 0s at the right of \(x_1\). Since the internal representation of numbers is in binary format, this operation is equivalent to multiplying \(x_1\) by \(2^{x_2}\).

**Parameters**

- **x1**: array_like of integer type  
  Input values.
- **x2**: array_like of integer type  
  Number of zeros to append to \(x_1\).
Returns

    out : array of integer type

    Return x1 with bits shifted x2 times to the left.

See Also:

    right_shift
    Shift the bits of an integer to the right.

    binary_repr
    Return the binary representation of the input number as a string.

Examples

    >>> np.binary_repr(5)
    '101'
    >>> np.left_shift(5, 2)
    20
    >>> np.binary_repr(20)
    '10100'
    >>> np.left_shift(5, [1,2,3])
    array([10, 20, 40])

right_shift()
Shift the bits of an integer to the right.

    Bits are shifted to the right by removing x2 bits at the right of x1. Since the internal representation of numbers is in binary format, this operation is equivalent to dividing x1 by 2 ** x2.

Parameters

    x1 : array_like, int
        Input values.

    x2 : array_like, int
        Number of bits to remove at the right of x1.

Returns

    out : ndarray, int
        Return x1 with bits shifted x2 times to the right.

See Also:

    left_shift
    Shift the bits of an integer to the left.

    binary_repr
    Return the binary representation of the input number as a string.

Examples

    >>> np.binary_repr(10)
    '1010'
    >>> np.right_shift(10, 1)
    5
3.11.2 Bit packing

packbits (myarray[, axis])
Packs the elements of a binary-valued array into bits in a uint8 array.

unpackbits (myarray[, axis])
Unpacks elements of a uint8 array into a binary-valued output array.

packbits ()
Packs the elements of a binary-valued array into bits in a uint8 array.
The result is padded to full bytes by inserting zero bits at the end.

Parameters

myarray : array_like
An integer type array whose elements should be packed to bits.

axis : int, optional
The dimension over which bit-packing is done. None implies packing the flattened array.

Returns

packed : ndarray
Array of type uint8 whose elements represent bits corresponding to the logical (0 or nonzero) value of the input elements. The shape of packed has the same number of dimensions as the input (unless axis is None, in which case the output is 1-D).

See Also:

unpackbits
Unpacks elements of a uint8 array into a binary-valued output array.

Examples

```python
>>> a = np.array([[1,0,1],
...                [0,1,0],
...                [1,1,0],
...                [0,0,1]])
>>> b = np.packbits(a, axis=-1)
>>> b
array([[160], [ 64], [192], [ 32]], dtype=uint8)
```

Note that in binary 160 = 1010 0000, 64 = 0100 0000, 192 = 1100 0000, and 32 = 0010 0000.

unpackbits ()
Unpacks elements of a uint8 array into a binary-valued output array.

Each element of myarray represents a bit-field that should be unpacked into a binary-valued output array. The shape of the output array is either 1-D (if axis is None) or the same shape as the input array with unpacking done along the axis specified.
Parameters

**myarray**: ndarray, uint8 type

Input array.

**axis**: int, optional

Unpacks along this axis.

Returns

**unpacked**: ndarray, uint8 type

The elements are binary-valued (0 or 1).

See Also:

**packbits**

Packs the elements of a binary-valued array into bits in a uint8 array.

Examples

```python
>>> a = np.array([[2], [7], [23]], dtype=np.uint8)
>>> a
array([[ 2],
       [ 7],
       [23]], dtype=uint8)
>>> b = np.unpackbits(a, axis=1)
>>> b
array([[0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 1, 0, 1, 1, 1]], dtype=uint8)
```

3.11.3 Output formatting

**binary_repr**(num[, width])  Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two’s complement of the number is returned, with respect to that width.

In a two’s-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers \[R23\]. A N-bit two’s-complement system can represent every integer in the range \(-2^{N-1} to +2^{N-1} - 1\).

Parameters

**num**: int

Only an integer decimal number can be used.

**width**: int, optional

The length of the returned string if num is positive, the length of the two’s complement if num is negative.

Returns

**bin**: str

Binary representation of num or two’s complement of num.
See Also:

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

Notes

binary_repr is equivalent to using base_repr with base 2, but about 25x faster.

References
[R23]

Examples

```python
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

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

```python
>>> np.binary_repr(-3, width=4)
'1101'
```

3.12 Statistics

3.12.1 Extremal values

amin (a[, axis, out])
Return the minimum along an axis.

amax (a[, axis, out])
Return the maximum along an axis.

nanmax (a[, axis])
Return the maximum of array elements over the given axis ignoring any NaNs.

nanmin (a[, axis])
Return the minimum of array elements over the given axis ignoring any NaNs.

ptp (a[, axis, out])
Range of values (maximum - minimum) along an axis.

Parameters

- **a**: array_like
  Input data.
- **axis**: int, optional
  Axis along which to operate. By default a flattened input is used.
- **out**: ndarray, optional
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See doc.ufuncs (Section “Output arguments”) for more details.

Returns

- **amin**: ndarray
  Return the minimum along an axis.
A new array or a scalar with the result, or a reference to `out` if it was specified.

**See Also:**

- `nanmin`  
  nan values are ignored instead of being propagated

- `fmin`  
  same behavior as the C99 `fmin` function

- `argmin`  
  Return the indices of the minimum values.

- `amax`, `nanmax`, `fmax`

**Notes**

NaN values are propagated, that is if at least one item is nan, the corresponding min value will be nan as well. To ignore NaN values (matlab behavior), please use `nanmin`.

**Examples**

```python
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)  # Minimum of the flattened array
0
>>> np.amin(a, axis=0)  # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1)  # Minima along the second axis
array([0, 2])

>>> b = np.arange(5, dtype=np.float)
>>> np.amin(b)
nan
>>> np.nanmin(b)
0.0
```

**amax** *(a, axis=None, out=None)*  
Return the maximum along an axis.

**Parameters**

- `a : array_like`  
  Input data.

- `axis : int, optional`  
  Axis along which to operate. By default flattened input is used.

- `out : ndarray, optional`  
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See `doc.ufuncs` (Section “Output arguments”) for more details.

**Returns**

- `amax : ndarray`
A new array or a scalar with the result, or a reference to `out` if it was specified.

**See Also:**

- `nanmax`  
  nan values are ignored instead of being propagated
- `fmax`  
  same behavior as the C99 fmax function
- `argmax`  
  Indices of the maximum values.

**Notes**

NaN values are propagated, that is if at least one item is nan, the corresponding max value will be nan as well. To ignore NaN values (matlab behavior), please use nanmax.

**Examples**

```python
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]]))
>>> np.amax(a)
3
>>> np.amax(a, axis=0)
array([2, 3])
>>> np.amax(a, axis=1)
array([1, 3])

>>> b = np.arange(5, dtype=np.float)
>>> np.amax(b)
nan
>>> np.nanmax(b)
4.0
```

**nanmax (a, axis=None)**

Return the maximum of array elements over the given axis ignoring any NaNs.

**Parameters**

- `a` : array_like  
  Array containing numbers whose maximum is desired. If `a` is not an array, a conversion is attempted.
- `axis` : int, optional  
  Axis along which the maximum is computed. The default is to compute the maximum of the flattened array.

**Returns**

- `y` : ndarray  
  An array with the same shape as `a`, with the specified axis removed. If `a` is a 0-d array, or if `axis` is `None`, a scalar is returned. The the same dtype as `a` is returned.

**See Also:**
**numpy.amax**
Maximum across array including any Not a Numbers.

**numpy.nanmin**
Minimum across array ignoring any Not a Numbers.

**isnan**
Shows which elements are Not a Number (NaN).

**isfinite**
Shows which elements are not: Not a Number, positive and negative infinity

**Notes**
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type, an integer type is returned unless the input contains NaNs and infinity.

**Examples**

```python
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmax(a)
3.0
>>> np.nanmax(a, axis=0)
array([ 3.,  2.])
>>> np.nanmax(a, axis=1)
array([ 2.,  3.])
```

When positive infinity and negative infinity are present:

```python
>>> np.nanmax([1, 2, np.nan, np.NINF])
2.0
>>> np.nanmax([1, 2, np.nan, np.inf])
inf
```

**nanmin**(a, axis=None)
Return the minimum of array elements over the given axis ignoring any NaNs.

**Parameters**
a : array_like
   Array containing numbers whose sum is desired. If a is not an array, a conversion is attempted.

axis : int, optional
   Axis along which the minimum is computed. The default is to compute the minimum of the flattened array.

Returns

y : ndarray or scalar
   An array with the same shape as a, with the specified axis removed. If a is a 0-d array, or if axis is None, a scalar is returned. The the same dtype as a is returned.

**See Also:**

**numpy.amin**
Minimum across array including any Not a Numbers.
**numpy.nanmax**
Maximum across array ignoring any Not a Numbers.

**isnan**
Shows which elements are Not a Number (NaN).

**isfinite**
Shows which elements are not: Not a Number, positive and negative infinity

**Notes**
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type, an integer type is returned unless the input contains NaNs and infinity.

**Examples**

```python
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmin(a)
1.0
>>> np.nanmin(a, axis=0)
array([ 1., 2.])
>>> np.nanmin(a, axis=1)
array([ 1., 3.])
```

When positive infinity and negative infinity are present:

```python
>>> np.nanmin([1, 2, np.nan, np.inf])
1.0
>>> np.nanmin([1, 2, np.nan, np.NINF])
-inf
```

**ptp** *(a, axis=None, out=None)*
Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

**Parameters**
- **a** : array_like
  Input values.
- **axis** : int, optional
  Axis along which to find the peaks. By default, flatten the array.
- **out** : array_like
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type of the output values will be cast if necessary.

**Returns**
- **ptp** : ndarray
  A new array holding the result, unless *out* was specified, in which case a reference to *out* is returned.
Examples

```python
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])

>>> np.ptp(x, axis=0)
array([2, 2])

>>> np.ptp(x, axis=1)
array([1, 1])
```

3.12.2 Averages and variances

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Compute the weighted average along the specified axis.</td>
</tr>
<tr>
<td>mean</td>
<td>Compute the arithmetic mean along the specified axis.</td>
</tr>
<tr>
<td>median</td>
<td>Compute the median along the specified axis.</td>
</tr>
<tr>
<td>std</td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td>var</td>
<td>Compute the variance along the specified axis.</td>
</tr>
</tbody>
</table>

**average** *(a, axis=None, weights=None, returned=False)*

Compute the weighted average along the specified axis.

**Parameters**

- **a**: array_like
  - Array containing data to be averaged. If `a` is not an array, a conversion is attempted.
- **axis**: int, optional
  - Axis along which to average `a`. If `None`, averaging is done over the flattened array.
- **weights**: array_like, optional
  - An array of weights associated with the values in `a`. Each value in `a` contributes to the average according to its associated weight. The weights array can either be 1-D (in which case its length must be the size of `a` along the given axis) or of the same shape as `a`. If `weights=None`, then all data in `a` are assumed to have a weight equal to one.
- **returned**: bool, optional
  - Default is `False`. If `True`, the tuple `(average, sum_of_weights)` is returned, otherwise only the average is returned. If `weights=None`, `sum_of_weights` is equivalent to the number of elements over which the average is taken.

**Returns**

- **average, [sum_of_weights]**: {array_type, double}
  - Return the average along the specified axis. When returned is `True`, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is `Float` if `a` is of integer type, otherwise it is of the same type as `a`. `sum_of_weights` is of the same type as `average`.

**Raises**

- ZeroDivisionError
  - When all weights along axis are zero. See `numpy.ma.average` for a version robust to this type of error.
**TypeError**:

When the length of 1D `weights` is not the same as the shape of `a` along axis.

See Also:

`mean`

`ma.average`

average for masked arrays

**Examples**

```python
g data = range(1,5)
g data
t 1, 2, 3, 4
g np.average(data)
2.5
g np.average(range(1,11), weights=range(10,0,-1))
4.0

g data = np.arange(6).reshape((3,2))
g data
array([[0, 1],
  [2, 3],
  [4, 5]])
g np.average(data, axis=1, weights=[1./4, 3./4])
array([ 0.75, 2.75, 4.75])
g np.average(data, weights=[1./4, 3./4])
Traceback (most recent call last):
  ...TypeError: Axis must be specified when shapes of a and weights differ.
```

**mean** (`a`, `axis=None`, `dtype=None`, `out=None`)

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

**Parameters**

- **a**: array_like
  
  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.

- **axis**: int, optional
  
  Axis along which the means are computed. The default is to compute the mean of the flattened array.

- **dtype**: dtype, optional
  
  Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

- **out**: ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary. See `doc.ufuncs` for details.

**Returns**

- **m**: ndarray, see dtype parameter above
If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See Also:**

`average`

Weighted average

**Notes**

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

In single precision, `mean` can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.546875
```

Computing the mean in float64 is more accurate:

```python
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

**median**

(a, axis=None, out=None, overwrite_input=False)

Compute the median along the specified axis.

Returns the median of the array elements.

**Parameters**

- `a` : array_like
  - Input array or object that can be converted to an array.

- `axis` : {None, int}, optional
  - Axis along which the medians are computed. The default (axis=None) is to compute the median along a flattened version of the array.

- `out` : ndarray, optional
  - Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

- `overwrite_input` : {False, True}, optional
If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is True and the input is not already an ndarray, an error will be raised.

**Returns**

median : ndarray

A new array holding the result (unless out is specified, in which case that array is returned instead). If the input contains integers, or floats of smaller precision than 64, then the output data-type is float64. Otherwise, the output data-type is the same as that of the input.

**See Also:**

mean

**Notes**

Given a vector V of length N, the median of V is the middle value of a sorted copy of V, \( V_{\text{sorted}} \), i.e., \( V_{\text{sorted}}[(N-1)/2] \), when N is odd. When N is even, it is the average of the two middle values of \( V_{\text{sorted}} \).

**Examples**

```python
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.median(a)
3.5
>>> np.median(a, axis=0)
array([ 6.5,  4.5,  2.5])
>>> np.median(a, axis=1)
array([ 7. ,  2. ])
>>> m = np.median(a, axis=0)
>>> out = np.zeros_like(m)
>>> np.median(a, axis=0, out=m)
array([ 6.5,  4.5,  2.5])
>>> m
array([ 6.5,  4.5,  2.5])
>>> b = a.copy()
>>> np.median(b, axis=1, overwrite_input=True)
array([ 7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.median(b, axis=None, overwrite_input=True)
3.5
```

**std**

\( a, \text{axis} = \text{None}, \text{dtype} = \text{None}, \text{out} = \text{None}, \text{ddof} = 0 \)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

a : array_like
Calculate the standard deviation of these values.

**axis**: int, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

**dtype**: dtype, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

**ddof**: int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is \( N - \text{ddof} \), where \( N \) represents the number of elements. By default \( \text{ddof} = 0 \).

**Returns**

- **standard_deviation**: ndarray, see dtype parameter above.

  If \( \text{out} \) is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

**See Also:**

- `var`, `mean`
- `numpy.doc.ufuncs`
  - Section “Output arguments”

**Notes**

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - \text{x.mean()}^2))} \).

The mean is normally calculated as \( x\cdot\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of the infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( \text{ddof}=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( \text{std} \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( \text{std} \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([1., 1.])
>>> np.std(a, axis=1)
array([0.5, 0.5])
```
In single precision, std() can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

```python
var (a, axis=None, dtype=None, out=None, ddof=0)
Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed
for the flattened array by default, otherwise over the specified axis.

Parameters

- a : array_like
  Array containing numbers whose variance is desired. If a is not an array, a conversion
  is attempted.

- axis : int, optional
  Axis along which the variance is computed. The default is to compute the variance of
  the flattened array.

- dtype : dtype, optional
  Type to use in computing the variance. For arrays of integer type the default is float32;
  for arrays of float types it is the same as the array type.

- out : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the
  expected output but the type is cast if necessary.

- ddof : int, optional
  “Delta Degrees of Freedom”: the divisor used in calculation is N – ddof, where N
  represents the number of elements. By default ddof is zero.

Returns

- variance : ndarray, see dtype parameter above
  If out=None, returns a new array containing the variance; otherwise a reference to the
  output array is returned.

See Also:

- std
  Standard deviation

- mean
  Average

numpy.doc.ufuncs
  Section “Output arguments”
```
Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x.\text{mean()}**) 2) \).

The mean is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of the infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a,0)
array([ 1., 1.])
>>> np.var(a,1)
array([ 0.25, 0.25])
```

In single precision, \( \text{var}() \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

3.12.3 Correlating

corrcoef(x[, y, rowvar, bias])
correlate(a, v[, mode, old_behavior])
cov(m[, y, rowvar, bias])

Return correlation coefficients.
Discrete, linear correlation of two 1-dimensional sequences.
Estimate a covariance matrix, given data.

```python
corrcoef(x, y=None, rowvar=1, bias=0)
Return correlation coefficients.
```

Please refer to the documentation for \( \text{cov} \) for more detail. The relationship between the correlation coefficient matrix, \( P \), and the covariance matrix, \( C \), is

\[
P_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} \cdot C_{jj}}}
\]
The values of P are between -1 and 1.

**Parameters**

- **m**: array_like
  A 1-D or 2-D array containing multiple variables and observations. Each row of m represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.

- **y**: array_like, optional
  An additional set of variables and observations. y has the same shape as m.

- **rowvar**: int, optional
  If `rowvar` is non-zero (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

- **bias**: int, optional
  Default normalization is by $(N-1)$, where N is the number of observations given (unbiased estimate). If `bias` is 1, then normalization is by $N$.

**Returns**

- **out**: ndarray
  The correlation coefficient matrix of the variables.

**See Also:**

- **cov**
  Covariance matrix

- **correlate** *(a, v, mode='valid', old_behavior=True)*
  Discrete, linear correlation of two 1-dimensional sequences.

  This function is equivalent to

  ```
  >>> np.correlate(a, v[::-1], mode=mode)
  ```

  where `v[::-1]` is the reverse of v.

**Parameters**

- **a, v**: array_like
  Input sequences.

- **mode**: {'valid', 'same', 'full'}, optional
  Refer to the `convolve` docstring. Note that the default is `valid`, unlike `convolve`, which uses `full`.

- **old_behavior**: bool
  If True, uses the old, numeric behavior (correlate(a,v) == correlate(v, a), and the conjugate is not taken for complex arrays). If False, uses the conventional signal processing definition (see note).

**See Also:**

- **convolve**
  Discrete, linear convolution of two one-dimensional sequences.
**acorrelate**
Discrete correlation following the usual signal processing definition for complex arrays, and without assuming that \( \text{correlate}(a, b) == \text{correlate}(b, a) \).

**Notes**
If `old_behavior` is False, this function computes the correlation as generally defined in signal processing texts:

\[
z[k] = \sum_n a[n] \ast \text{conj}(v[n+k])
\]

with \( a \) and \( v \) sequences being zero-padded where necessary and conj being the conjugate.

**Examples**

```python
>>> np.correlate([1, 2, 3], [0, 1, 0.5])
anarray([ 3.5])
```

```python
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "same")
anarray([ 2. , 3.5, 3. ])
```

```python
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "full")
anarray([ 0.5, 2. , 3.5, 3. , 0. ])
```

**cov** *(m, y=None, rowvar=1, bias=0)*
Estimate a covariance matrix, given data.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, \( X = [x_1, x_2, \ldots, x_N]^T \), then the covariance matrix element \( C_{ij} \) is the covariance of \( x_i \) and \( x_j \). The element \( C_{ii} \) is the variance of \( x_i \).

**Parameters**
- **m**: array_like
  A 1-D or 2-D array containing multiple variables and observations. Each row of \( m \) represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.
- **y**: array_like, optional
  An additional set of variables and observations. \( y \) has the same form as that of \( m \).
- **rowvar**: int, optional
  If `rowvar` is non-zero (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.
- **bias**: int, optional
  Default normalization is by \((N-1)\), where \( N \) is the number of observations given (unbiased estimate). If `bias` is 1, then normalization is by \( N \).

**Returns**
- **out**: ndarray
  The covariance matrix of the variables.

**See Also:**
- `corrcoef`
  Normalized covariance matrix
**Examples**

Consider two variables, \(x_0\) and \(x_1\), which correlate perfectly, but in opposite directions:

```python
>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
```

Note how \(x_0\) increases while \(x_1\) decreases. The covariance matrix shows this clearly:

```python
>>> np.cov(x)
array([[ 1., -1.],
       [-1., 1.]])
```

Note that element \(C_{0,1}\), which shows the correlation between \(x_0\) and \(x_1\), is negative.

Further, note how \(x\) and \(y\) are combined:

```python
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = np.vstack((x,y))
>>> print np.cov(X)
[[ 11.71 -4.286 ]
 [ -4.286 2.14413333]]
>>> print np.cov(x, y)
[[ 11.71 -4.286 ]
 [ -4.286 2.14413333]]
>>> print np.cov(x)
11.71
```

### 3.12.4 Histograms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>Compute the histogram of a set of data.</td>
</tr>
<tr>
<td>histogram2d</td>
<td>Compute the bi-dimensional histogram of two data samples.</td>
</tr>
<tr>
<td>histogramdd</td>
<td>Compute the multidimensional histogram of some data.</td>
</tr>
<tr>
<td>bincount</td>
<td>Count number of occurrences of each value in array of non-negative ints.</td>
</tr>
<tr>
<td>digitize</td>
<td>Return the indices of the bins to which each value in input array belongs.</td>
</tr>
</tbody>
</table>

**histogram**

Computes the histogram of a set of data.

**Parameters**

- **a**: array_like
  - Input data. The histogram is computed over the flattened array.
- **bins**: int or sequence of scalars, optional
  - If `bins` is an int, it defines the number of equal-width bins in the given range (10, by default). If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.
- **range**: (float, float), optional
The lower and upper range of the bins. If not provided, range is simply \((a.\text{min}(), a.\text{max}())\). Values outside the range are ignored. Note that with new set to False, values below the range are ignored, while those above the range are tallied in the rightmost bin.

normed : bool, optional

If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Note that the sum of the histogram values will not be equal to 1 unless bins of unity width are chosen; it is not a probability mass function.

weights : array_like, optional

An array of weights, of the same shape as \(a\). Each value in \(a\) only contributes its associated weight towards the bin count (instead of 1). If normed is True, the weights are normalized, so that the integral of the density over the range remains 1. The weights keyword is only available with new set to True.

new : {None, True, False}, optional

Whether to use the new semantics for histogram:

- None : the new behaviour is used, no warning is printed.
- True : the new behaviour is used and a warning is raised about the future removal of the new keyword.
- False : the old behaviour is used and a DeprecationWarning is raised.

As of NumPy 1.3, this keyword should not be used explicitly since it will disappear in NumPy 2.0.

Returns

hist : array

The values of the histogram. See normed and weights for a description of the possible semantics.

bin_edges : array of dtype float

Return the bin edges \((\text{length(hist)}+1)\). With new=False, return the left bin edges \((\text{length(hist)})\).

See Also:

histogramdd, bincount, searchsorted

Notes

All but the last (righthand-most) bin is half-open. In other words, if bins is:

\([1, 2, 3, 4]\)

then the first bin is \([1, 2)\) (including 1, but excluding 2) and the second \([2, 3)\). The last bin, however, is \([3, 4]\), which includes 4.

Examples

```python
>>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
(array([0, 2, 1]), array([0, 1, 2, 3]))

>>> np.histogram(np.arange(4), bins=np.arange(5), normed=True)
```
```python
(array([0.25, 0.25, 0.25, 0.25]), array([0, 1, 2, 3, 4]))
>>> np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3])
(array([1, 4, 1]), array([0, 1, 2, 3]))
```

```python
>> a = np.arange(5)
>>> hist, bin_edges = np.histogram(a, normed=True)
>>> hist
array([0.5, 0. , 0.5, 0. , 0.5, 0. , 0.5, 0. , 0.5, 0.5])
>>> hist.sum()
2.4999999999999996
>>> np.sum(hist*np.diff(bin_edges))
1.0
```

**histogram2d**

```python
(x, y, bins=10, range=None, normed=False, weights=None)
```

Compute the bi-dimensional histogram of two data samples.

**Parameters**

- **x**: array_like, shape(N,)
  - A sequence of values to be histogrammed along the first dimension.

- **y**: array_like, shape(M,)
  - A sequence of values to be histogrammed along the second dimension.

- **bins**: int or [int, int] or array_like or [array, array], optional
  - The bin specification:
    - If int, the number of bins for the two dimensions (nx=ny=bins).
    - If [int, int], the number of bins in each dimension (nx, ny = bins).
    - If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
    - If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

- **range**: array_like, shape(2,2), optional
  - The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the **bins** parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

- **normed**: bool, optional
  - If False, returns the number of samples in each bin. If True, returns the bin density, i.e. the bin count divided by the bin area.

- **weights**: array_like, shape(N,), optional
  - An array of values w_i weighing each sample (x_i, y_i). Weights are normalized to 1 if **normed** is True. If **normed** is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

**Returns**

- **H**: ndarray, shape(nx, ny)
  - The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

- **xedges**: ndarray, shape(nx,)
  - The bin edges along the first dimension.
yedges : ndarray, shape(ny,)

The bin edges along the second dimension.

See Also:

histogram
1D histogram

histogramdd
Multidimensional histogram

Notes

When normed is True, then the returned histogram is the sample density, defined such that:

$$\sum_{i=0}^{nx-1} \sum_{j=0}^{ny-1} H_{i,j} \Delta x_i \Delta y_j = 1$$

where $H$ is the histogram array and $\Delta x_i \Delta y_j$ the area of bin $[i,j]$.

Please note that the histogram does not follow the Cartesian convention where $x$ values are on the abcissa and $y$ values on the ordinate axis. Rather, $x$ is histogrammed along the first dimension of the array (vertical), and $y$ along the second dimension of the array (horizontal). This ensures compatibility with histogramdd.

Examples

```python
>>> x, y = np.random.randn(2, 100)
>>> H, xedges, yedges = np.histogram2d(x, y, bins=(5, 8))
>>> H.shape, xedges.shape, yedges.shape
((5,8), (6,), (9,))

We can now use the Matplotlib to visualize this 2-dimensional histogram:

```
**histogramdd** (*sample, bins=10, range=None, normed=False, weights=None*)

Compute the multidimensional histogram of some data.

**Parameters**

- **sample** : array_like
  The data to be histogrammed. It must be an (N,D) array or data that can be converted to such. The rows of the resulting array are the coordinates of points in a D dimensional polytope.

- **bins** : sequence or int, optional
  The bin specification:
  - A sequence of arrays describing the bin edges along each dimension.
  - The number of bins for each dimension (nx, ny, ... =bins)
  - The number of bins for all dimensions (nx=ny=...=bins).

- **range** : sequence, optional
  A sequence of lower and upper bin edges to be used if the edges are not given explicitly in *bins*. Defaults to the minimum and maximum values along each dimension.

- **normed** : boolean, optional
  If False, returns the number of samples in each bin. If True, returns the bin density, ie, the bin count divided by the bin hypervolume.

- **weights** : array_like (N,), optional
  An array of values w_j weighing each sample (x_i, y_j, z_i, ...). Weights are normalized to 1 if normed is True. If normed is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

**Returns**

- **H** : ndarray
  The multidimensional histogram of sample x. See normed and weights for the different possible semantics.

- **edges** : list
  Parameters
A list of D arrays describing the bin edges for each dimension.

See Also:

- `histogram`  
  1D histogram
- `histogram2d`  
  2D histogram

Examples

```python
>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5,8,4), 6, 9, 5)
```

`bincount()`  
Count number of occurrences of each value in array of non-negative ints.

The number of bins (of size 1) is one larger than the largest value in `x`. Each bin gives the number of occurrences of its index value in `x`. If `weights` is specified the input array is weighted by it, i.e. if a value `n` is found at position `i`, `out[n] += weight[i]` instead of `out[n] += 1`.

Parameters

- `x`: array_like, 1 dimension, nonnegative ints  
  Input array.
- `weights`: array_like, optional  
  Weights, array of the same shape as `x`.

Returns

- `out`: ndarray of ints  
  The result of binning the input array. The length of `out` is equal to `np.amax(x)+1`.

Raises

- `ValueError`:  
  If the input is not 1-dimensional, or contains elements with negative values.
- `TypeError`:  
  If the type of the input is float or complex.

See Also:

- `histogram`, `digitize`, `unique`

Examples

```python
>>> np.bincount(np.arange(5))
array([1, 1, 1, 1, 1])
>>> np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
array([1, 3, 1, 1, 0, 0, 0, 1])
```

```python
>>> x = np.array([0, 1, 1, 3, 2, 1, 7, 23])
>>> np.bincount(x).size == np.amax(x)+1
True
```
```python
>>> np.bincount(np.arange(5, dtype=np.float))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: array cannot be safely cast to required type
```

A possible use of `bincount` is to perform sums over variable-size chunks of an array, using the `weights` keyword.

```python
>>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6])  # weights
>>> x = np.array([0, 1, 1, 2, 2, 2])
>>> np.bincount(x, weights=w)
array([ 0.3, 0.7, 1.1])
```

digitize()

Return the indices of the bins to which each value in input array belongs.

Each index i returned is such that `bins[i-1] <= x < bins[i]` if `bins` is monotonically increasing, or `bins[i-1] > x >= bins[i]` if `bins` is monotonically decreasing. If values in `x` are beyond the bounds of `bins`, 0 or `len(bins)` is returned as appropriate.

**Parameters**
- `x`: array_like
  Input array to be binned. It has to be 1-dimensional.
- `bins`: array_like
  Array of bins. It has to be 1-dimensional and monotonic.

**Returns**
- `out`: ndarray of ints
  Output array of indices, of same shape as `x`.

**Raises**
- `ValueError`
  If the input is not 1-dimensional, or if `bins` is not monotonic.
- `TypeError`
  If the type of the input is complex.

**See Also:**
- bincount, histogram, unique

**Notes**
If values in `x` are such that they fall outside the bin range, attempting to index `bins` with the indices that `digitize` returns will result in an IndexError.

**Examples**

```python
>>> x = np.array([0.2, 6.4, 3.0, 1.6])
>>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = np.digitize(x, bins)
>>> inds
array([1, 4, 3, 2])
>>> for n in range(x.size):
...     print bins[inds[n]-1], "<=", x[n], "<", bins[inds[n]]
0.00 <= 0.20 < 1.00
1.00 <= 3.00 < 2.50
2.50 <= 6.40 < 4.00
1.00 <= 1.60 < 2.50
```
3.13 Mathematical functions

3.13.1 Trigonometric functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(x, out)</td>
<td>Trigonometric sine, element-wise.</td>
</tr>
<tr>
<td>cos(x, out)</td>
<td>Cosine elementwise.</td>
</tr>
<tr>
<td>tan(x, out)</td>
<td>Compute tangent element-wise.</td>
</tr>
<tr>
<td>arcsin(x, out)</td>
<td>Inverse sine elementwise.</td>
</tr>
<tr>
<td>arccos(x, out)</td>
<td>Trigonometric inverse cosine, element-wise.</td>
</tr>
<tr>
<td>arctan(x, out)</td>
<td>Trigonometric inverse tangent, element-wise.</td>
</tr>
<tr>
<td>hypot(x1, x2, out)</td>
<td>Given the “legs” of a right triangle, return its hypotenuse.</td>
</tr>
<tr>
<td>arctan2(x1, x2, out)</td>
<td>Elementwise arc tangent of x1/x2 choosing the quadrant correctly.</td>
</tr>
<tr>
<td>degrees(x, out)</td>
<td>Convert angles from radians to degrees.</td>
</tr>
<tr>
<td>radians(x, out)</td>
<td>Convert angles from degrees to radians.</td>
</tr>
<tr>
<td>unwrap(p, discont, axis)</td>
<td>Unwrap by changing deltas between values to 2*pi complement.</td>
</tr>
<tr>
<td>deg2rad(x, out)</td>
<td>Convert angles from degrees to radians.</td>
</tr>
<tr>
<td>rad2deg(x, out)</td>
<td>Convert angles from radians to degrees.</td>
</tr>
</tbody>
</table>

**sin()**

Trigonometric sine, element-wise.

**Parameters**

- x : array_like
  Angle, in radians (2π rad equals 360 degrees).

**Returns**

- y : array_like
  The sine of each element of x.

**See Also:**

arcsin, sinh, cos

**Notes**

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the +x axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray’s intersection with the unit circle is the sine of that angle. It ranges from -1 for x = 3π/2 to +1 for π/2. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

**Examples**

Print sine of one angle:

```python
>>> np.sin(np.pi/2)
1.0
```
Print sines of an array of angles given in degrees:

```python
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0. , 0.5 , 0.70710678, 0.8660254 , 1. ])
```

Plot the sine function:

```python
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

The corresponding cosine values.

```python
cos()
```

**Parameters**

- `x` : array_like
  
  Input array in radians.

- `out` : ndarray, optional
  
  Output array of same shape as `x`.

**Returns**

- `y` : ndarray
  
  The corresponding cosine values.

**Raises**

- `ValueError: invalid return array shape`
  
  if `out` is provided and `out.shape != x.shape` (See Examples)

**Notes**

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)
References


Examples

```python
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([ 1.00000000e+00, 6.12303177e-17, -1.00000000e+00])

# Example of providing the optional output parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True

# Example of ValueError due to provision of shape mis-matched 'out'
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
ValueError: invalid return array shape
```

tan()

Compute tangent element-wise.

Equivalent to \( \frac{\sin(x)}{\cos(x)} \) element-wise.

Parameters

- **x**: array_like
  
  Input array.

- **out**: ndarray, optional
  
  Output array of same shape as \( x \).

Returns

- **y**: ndarray
  
  The corresponding tangent values.

 Raises

- **ValueError**: invalid return array shape :
  
  if `out` is provided and `out.shape != x.shape` (See Examples)

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References


Examples

```python
>>> from math import pi
>>> np.tan(np.array([-pi, pi/2, pi]))
array([ 1.22460635e-16, 1.63317787e+16, -1.22460635e-16])

# Example of providing the optional output parameter illustrating
# that what is returned is a reference to said parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
```
True

>>> # Example of ValueError due to provision of shape mis-matched 'out'
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape

arcsin()
Inverse sine elementwise.

Parameters
  x : array_like
      y-coordinate on the unit circle.

  out : ndarray, optional
      Array of the same shape as x, to store results in. See doc.ufuncs (Section “Output
      arguments”) for more details.

Returns
  angle : ndarray
      The angle of the ray intersecting the unit circle at the given y-coordinate in radians
      [-pi, pi]. If x is a scalar then a scalar is returned, otherwise an array is returned.

See Also:
sin, cos, arccos, tan, arctan, arctan2, emath.arcsin

Notes

arcsin is a multivalued function: for each x there are infinitely many numbers z such that
sin(z) = x. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, arcsin always returns real output. For each value that cannot be expressed as a
real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, arcsin is a complex analytical function that has branch cuts [-inf, -1] and [1, inf] and
is continuous from above on the former and from below on the latter.

The inverse sine is also known as asin or sin^-1.

References

[R6], [R7]

Examples

>>> np.arcsin(1)  # pi/2
1.5707963267948966
>>> np.arcsin(-1) # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0

arccos()
Trigonometric inverse cosine, element-wise.

The inverse of cos so that, if y = cos(x), then x = arccos(y).
Parameters

\texttt{x} : array_like

\texttt{x}-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

\texttt{out} : ndarray, optional

Array of the same shape as \texttt{a}, to store results in. See \texttt{doc.ufuncs} (Section “Output arguments”) for more details.

Returns

\texttt{angle} : ndarray

The angle of the ray intersecting the unit circle at the given \texttt{x}-coordinate in radians [0, \pi]. If \texttt{x} is a scalar then a scalar is returned, otherwise an array of the same shape as \texttt{x} is returned.

See Also:

cos, arctan, arcsin, emath.arccos

Notes

\texttt{arccos} is a multivalued function: for each \texttt{x} there are infinitely many numbers \texttt{z} such that \texttt{cos(z)} = \texttt{x}. The convention is to return the angle \texttt{z} whose real part lies in [0, \pi].

For real-valued input data types, \texttt{arccos} always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \texttt{nan} and sets the invalid floating point error flag.

For complex-valued input, \texttt{arccos} is a complex analytical function that has branch cuts [-\texttt{inf}, -1] and [1, \texttt{inf}] and is continuous from above on the former and from below on the latter.

The inverse \texttt{cos} is also known as \texttt{acos} or cos^{-1}.

References

[R2], [R3]

Examples

We expect the \texttt{arccos} of 1 to be 0, and of -1 to be \pi:

\begin{verbatim}
>>> np.arccos([1, -1])
array([ 0. , 3.14159265])
\end{verbatim}

Plot \texttt{arccos}:

\begin{verbatim}
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis(‘tight’)  
>>> plt.show()
\end{verbatim}
arctan()  
Trigonometric inverse tangent, element-wise.  
The inverse of tan, so that if \( y = \tan(x) \) then \( x = \arctan(y) \).

Parameters  
- \( x \) : array_like  
  Input values. \( \arctan \) is applied to each element of \( x \).

Returns  
- \( out \) : ndarray  
  Out has the same shape as \( x \). Its real part is in \([-\pi/2, \pi/2]\). It is a scalar if \( x \) is a scalar.

See Also:  
arctan2  
Calculate the arctan of \( y/x \).

Notes  
arctan is a multivalued function: for each \( x \) there are infinitely many numbers \( z \) such that \( \tan(z) = x \). The convention is to return the angle \( z \) whose real part lies in \([-\pi/2, \pi/2]\).

For real-valued input data types, \( \arctan \) always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \( \text{nan} \) and sets the \text{invalid} floating point error flag.

For complex-valued input, \( \text{arctan} \) is a complex analytical function that has branch cuts \([1j, \infty j]\) and \([-1j, -\infty j]\) and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as \( \text{atan} \) or \( \tan^{-1} \).

References  
[R10], [R11]

Examples  
We expect the arctan of 0 to be 0, and of 1 to be \( \pi/4 \):
>>> np.arctan([0, 1])
array([0., 0.78539816])

>>> np.pi/4
0.78539816339744828

Plot arctan:

```python
>>> import matplotlib.pyplot as plt

>>> x = np.linspace(-10, 10)

>>> plt.plot(x, np.arctan(x))

>>> plt.axis('tight')

>>> plt.show()
```

---

**hypot()**

Given the “legs” of a right triangle, return its hypotenuse.

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If `x1` or `x2` is scalar_like (i.e., unambiguously castable to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

**Parameters**

- `x1, x2`: array_like
  - Leg of the triangle(s).

- `out`: ndarray, optional
  - Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

**Returns**

- `z`: ndarray
  - The hypotenuse of the triangle(s).
Examples

```python
>>> np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Example showing broadcast of scalar_like argument:

```python
>>> np.hypot(3*np.ones((3, 3)), [4])
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

```python
arctan2
```
Elementwise arc tangent of \(x_1/x_2\) choosing the quadrant correctly.

The quadrant (ie. branch) is chosen so that \(\text{arctan2}(x_1, x_2)\) is the signed angle in radians between the line segments \((0,0) - (1,0)\) and \((0,0) - (x_2,x_1)\). This function is defined also for \(x_2 = 0\).

\(\text{arctan2}\) is not defined for complex-valued arguments.

**Parameters**
- \(x1\) : array_like, real-valued
  y-coordinates.
- \(x2\) : array_like, real-valued
  x-coordinates. \(x2\) must be broadcastable to match the shape of \(x1\), or vice versa.

**Returns**
- \(angle\) : ndarray
  Array of angles in radians, in the range \([-\pi, \pi]\).

**See Also:**
- `arctan`, `tan`

**Notes**
\(\text{arctan2}\) is identical to the \(\text{atan2}\) function of the underlying C library. The following special values are defined in the C standard [2]:

<table>
<thead>
<tr>
<th>(x1)</th>
<th>(x2)</th>
<th>(\text{arctan2}(x1,x2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/- 0</td>
<td>+/- 0</td>
<td>+/- 0</td>
</tr>
<tr>
<td>+/- 0</td>
<td>-0</td>
<td>+/- pi</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>+/- inf</td>
<td>+0 / +pi</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>+/- inf</td>
<td>-0 / -pi</td>
</tr>
<tr>
<td>+/- inf</td>
<td>+inf</td>
<td>+/- (\pi/4)</td>
</tr>
<tr>
<td>+/- inf</td>
<td>-inf</td>
<td>+/- (3\pi/4)</td>
</tr>
</tbody>
</table>

Note that +0 and -0 are distinct floating point numbers.

**References**
- [R12], [R13]
**Examples**

Consider four points in different quadrants:

```python
>>> x = np.array([-1, +1, +1, -1])
>>> y = np.array([-1, -1, +1, +1])
>>> np.arctan2(y, x) * 180 / np.pi
array([-135., -45., 45., 135.])
```

Note the order of the parameters. \( \arctan2 \) is defined also when \( x_2 = 0 \) and at several other special points, obtaining values in the range \([-\pi, \pi]\):

```python
>>> np.arctan2([1., -1.], [0., 0.])
array([ 1.57079633, -1.57079633])
>>> np.arctan2([0., 0., np.inf], [+0., -0., np.inf])
array([ 0. , 3.14159265, 0.78539816])
```

degrees()

Convert angles from radians to degrees.

**Parameters**

- **x**: array_like
  - Input array in radians.

- **out**: ndarray, optional
  - Output array of same shape as x.

**Returns**

- **y**: ndarray of floats
  - The corresponding degree values.

**See Also:**

- rad2deg
  - equivalent function

**Examples**

Convert a radian array to degrees

```python
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([ 0., 30., 60., 90., 120., 150., 180., 210., 240.,
       270., 300., 330.])
```

```python
>>> out = np.zeros((rad.shape))
>>> r = degrees(rad, out)
>>> np.all(r == out)
True
```

degrees()

Convert angles from radians to degrees.

**Parameters**

- **x**: array_like
  - Input array in degrees.
out : ndarray, optional

Output array of same shape as x.

Returns
y : ndarray

The corresponding radian values.

See Also:
deg2rad
equivalent function

Examples
Convert a degree array to radians

```python
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0. , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265, 3.66519143, 4.1887902 , 4.71238898,
       5.23598776, 5.75958653])
```

```python
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

unwrap (p, discont=3.1415926535897931, axis=-1)

Unwrap by changing deltas between values to 2*pi complement.

Unwrap radian phase p by changing absolute jumps greater than discont to their 2*pi complement along the given axis.

Parameters
p : array_like

Input array.

discont : float, optional

Maximum discontinuity between values, default is \( \pi \).

axis : int, optional

Axis along which unwrap will operate, default is the last axis.

Returns
out : ndarray

Output array.

See Also:
rad2deg, deg2rad

Notes
If the discontinuity in p is smaller than \( \pi \), but larger than discont, no unwrapping is done because taking the 2*pi complement would only make the discontinuity larger.
Examples

```python
>>> phase = np.linspace(0, np.pi, num=5)
>>> phase[3:] += np.pi
>>> phase
array([ 0. , 0.78539816, 1.57079633, 5.49778714, 6.28318531])
>>> np.unwrap(phase)
array([ 0. , 0.78539816, 1.57079633, -0.78539816, 0. ])
```

deg2rad()

Convert angles from degrees to radians.

Parameters

- `x` : array_like
  Angles in degrees.

Returns

- `y` : ndarray
  The corresponding angle in radians.

See Also:

- `rad2deg`
  Convert angles from radians to degrees.
- `unwrap`
  Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0. deg2rad(x) is x * pi / 180.

Examples

```python
>>> np.deg2rad(180)
3.1415926535897931
```

rad2deg()

Convert angles from radians to degrees.

Parameters

- `x` : array_like
  Angle in radians.
- `out` : ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

- `y` : ndarray
  The corresponding angle in degrees.

See Also:

- `deg2rad`
  Convert angles from degrees to radians.

3.13. Mathematical functions
unwrap

Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0. rad2deg(x) is 180 * x / pi.

Examples

```python
>>> np.rad2deg(np.pi/2)
90.0
```

3.13.2 Hyperbolic functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinh</td>
<td>Hyperbolic sine, element-wise.</td>
</tr>
<tr>
<td>cosh</td>
<td>Hyperbolic cosine, element-wise.</td>
</tr>
<tr>
<td>tanh</td>
<td>Compute hyperbolic tangent element-wise.</td>
</tr>
<tr>
<td>arcsinh</td>
<td>Inverse hyperbolic sine elementwise.</td>
</tr>
<tr>
<td>arccosh</td>
<td>Inverse hyperbolic cosine, elementwise.</td>
</tr>
<tr>
<td>arctanh</td>
<td>Inverse hyperbolic tangent elementwise.</td>
</tr>
</tbody>
</table>

**sinh()**

Hyperbolic sine, element-wise.

Equivalent to 1/2 * (np.exp(x) - np.exp(-x)) or -1j * np.sin(1j*x).

**Parameters**

- x : array_like
  - Input array.
- out : ndarray, optional
  - Output array of same shape as x.

**Returns**

- y : ndarray
  - The corresponding hyperbolic sine values.

**Raises**

ValueError: invalid return array shape :

if out is provided and out.shape != x.shape (See Examples)

**Notes**

If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)

**References**


**Examples**

```python
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j)  # (exact value is 0)
```
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.

>>> # Example of providing the optional output parameter
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True

>>> # Example of ValueError due to provision of shape mis-matched 'out'
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
 ValueError: invalid return array shape

cosh()
Hyperbolic cosine, element-wise.

Equivalent to 1/2 * (np.exp(x) + np.exp(-x)) and np.cos(1j*x).

Parameters
x : array_like
   Input array.

Returns
out : ndarray
   Output array of same shape as x.

Examples

>>> np.cosh(0)
1.0

The hyperbolic cosine describes the shape of a hanging cable:

>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
tanh()
Compute hyperbolic tangent element-wise.

Equivalent to \( \frac{\sinh(x)}{\cosh(x)} \) or \(-1j \times \tan(1j \times x)\).

Parameters
- **x**: array_like
  Input array.
- **out**: ndarray, optional
  Output array of same shape as `x`.

Returns
- **y**: ndarray
  The corresponding hyperbolic tangent values.

Raises
- **ValueError**: invalid return array shape :
  if `out` is provided and `out.shape != x.shape` (See Examples)

Notes
If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References
[R140], [R141]

Examples

```python
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))
array([ 0. +0.00000000e+00j, 0. -1.22460635e-16j, 0. +1.63317787e+16j])
```

```python
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.tanh([0.1], out1)
```
arcsinh()  
Inverse hyperbolic sine elementwise.

Parameters
   x : array_like
      Input array.
   out : ndarray, optional
      Array into which the output is placed. Its type is preserved and it must be of the right
      shape to hold the output. See doc.ufuncs.

Returns
   out : ndarray
      Array of the same shape as x.

Notes
arcsinh is a multivalued function: for each x there are infinitely many numbers z such that sinh(z) = x. The
convention is to return the z whose imaginary part lies in [-pi/2, pi/2].

For real-valued input data types, arcsinh always returns real output. For each value that cannot be expressed as
a real number or infinity, it returns nan and sets the invalid floating point error flag.

For complex-valued input, arccos is a complex analytical function that has branch cuts [1j, infj] and [-1j, -infj]
and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as asinh or sinh^-1.

References
[R8], [R9]

Examples
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256, 2.99822295])

arccosh()  
Inverse hyperbolic cosine, elementwise.

Parameters
   x : array_like
      Input array.
   out : ndarray, optional
      Array of the same shape as x, to store results in. See doc.ufuncs (Section “Output
      arguments”) for details.
Returns

\[ y : \text{ndarray} \]

Array of the same shape as \( x \).

See Also:

cosh, arcsinh, sinh, arctanh, tanh

Notes

arccosh is a multivalued function: for each \( x \) there are infinitely many numbers \( z \) such that \( \cosh(z) = x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi, \pi]\) and the real part in \([0, \infty]\).

For real-valued input data types, arccosh always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, arccosh is a complex analytical function that has a branch cut \([-\infty, 1]\) and is continuous from above on it.

References

[R4], [R5]

Examples

```python
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445, 2.99322285])
```

>>>

arctanh()

Inverse hyperbolic tangent elementwise.

Parameters

\[ x : \text{array_like} \]

Input array.

Returns

\[ out : \text{ndarray} \]

Array of the same shape as \( x \).

See Also:

ecmath.arctanh

Notes

arctanh is a multivalued function: for each \( x \) there are infinitely many numbers \( z \) such that \( \tanh(z) = x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi/2, \pi/2]\).

For real-valued input data types, arctanh always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, arctanh is a complex analytical function that has branch cuts \([-1, -\infty]\) and \([1, \infty]\) and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as atanh or \( \tanh^{-1} \).

References

[R14], [R15]
Examples

```python
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

3.13.3 Rounding

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>around(a, decimals)</code>, <code>round_(a, decimals)</code>, <code>rint(x, out)</code>, <code>fix(x, y)</code>, <code>floor(x, out)</code>, <code>ceil(x, out)</code>, <code>trunc(x, out)</code></td>
<td>Evenly round to the given number of decimals. Round an array to the given number of decimals. Round elements of the array to the nearest integer. Round to nearest integer towards zero. Return the floor of the input, element-wise. Return the ceiling of the input, element-wise. Return the truncated value of the input, element-wise.</td>
</tr>
</tbody>
</table>

`around(a, decimals=0, out=None)`

Evenly round to the given number of decimals.

**Parameters**

- `a`: array_like
  - Input data.
- `decimals`: int, optional
  - Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.
- `out`: ndarray, optional
  - Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary. See `doc.ufuncs` (Section “Output arguments”) for details.

**Returns**

- `rounded_array`: ndarray
  - An array of the same type as `a`, containing the rounded values. Unless `out` was specified, a new array is created. A reference to the result is returned.

  The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

**See Also:**

- `ndarray.round`
  - equivalent method

**Notes**

For values exactly halfway between rounded decimal values, Numpy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in the IEEE floating point standard [R16] and errors introduced when scaling by powers of ten.
References

[R16], [R17]

Examples

```python
>>> np.around([0.37, 1.64])
array([ 0., 2.])
>>> np.around([0.37, 1.64], decimals=1)
array([ 0.4, 1.6])
>>> np.around([0.5, 1.5, 2.5, 3.5, 4.5])  # rounds to nearest even value
array([ 0., 2., 2., 4., 4.])
>>> np.around([1,2,3,11], decimals=1)  # ndarray of ints is returned
array([ 1, 2, 3, 11])
>>> np.around([1,2,3,11], decimals=-1)
array([ 0, 0, 0, 10])
```

`round_`(a, decimals=0, out=None)
Round an array to the given number of decimals.

Refer to `around` for full documentation.

See Also:

`around`
equivalent function

`rint`
Round elements of the array to the nearest integer.

Parameters

- `x` : array_like
  Input array.

Returns

- `out` : {ndarray, scalar}
  Output array is same shape and type as `x`.

See Also:

`ceil`, `floor`, `trunc`

Examples

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0., 0., 2., 2., 2.])
```

`fix`(x, y=None)
Round to nearest integer towards zero.

Round an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

Parameters

- `x` : array_like
  An array of floats to be rounded

- `y` : ndarray, optional

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0., 0., 2., 2., 2.])
```
Output array

Returns

out : ndarray of floats

The array of rounded numbers

See Also:

trunc, floor, ceil

around

Round to given number of decimals

Examples

>>> np.fix(3.14)
3.0
>>> np.fix(3)
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9])
array([ 2., 2., -2., -2.])

floor()

Return the floor of the input, element-wise.

The floor of the scalar $x$ is the largest integer $i$, such that $i$. It is often denoted as $\lfloor x \rfloor$.

Parameters

x : array_like

Input data.

Returns

y : {ndarray, scalar}

The floor of each element in x.

See Also:

ceil, trunc, rint

Notes

Some spreadsheet programs calculate the “floor-towards-zero”, in other words floor(-2.5) == -2. NumPy, however, uses the a definition of floor such that floor(-2.5) == -3.

Examples

>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.floor(a)
array([-2., -2., -1., 0., 1., 1., 2.])

ceil()

Return the ceiling of the input, element-wise.

The ceil of the scalar $x$ is the smallest integer $i$, such that $i >= x$. It is often denoted as $\lceil x \rceil$.

Parameters

x : array_like

Input data.
Returns

.. code-block:: python

    y : {ndarray, scalar}

    The ceiling of each element in `x`, with float dtype.

See Also:

floor, trunc, rint

Examples

>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.ceil(a)
array([-1., -1., -0., 1., 2., 2., 2.])

trunc()

Return the truncated value of the input, element-wise.

The truncated value of the scalar `x` is the nearest integer `i` which is closer to zero than `x` is. In short, the fractional part of the signed number `x` is discarded.

Parameters

.. code-block:: python

    x : array_like

        Input data.

Returns

.. code-block:: python

    y : {ndarray, scalar}

        The truncated value of each element in `x`.

See Also:

ceil, floor, rint

Notes

New in version 1.3.0.

Examples

>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.trunc(a)
array([-1., -1., -0., 0., 1., 1., 2.])

3.13.4 Sums, products, differences

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod</td>
<td>Return the product of array elements over a given axis.</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of array elements over a given axis.</td>
</tr>
<tr>
<td>nansum</td>
<td>Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.</td>
</tr>
<tr>
<td>cumprod</td>
<td>Return the cumulative product of elements along a given axis.</td>
</tr>
<tr>
<td>cumsum</td>
<td>Return the cumulative sum of the elements along a given axis.</td>
</tr>
<tr>
<td>diff</td>
<td>Calculate the n-th order discrete difference along given axis.</td>
</tr>
<tr>
<td>ediff1d</td>
<td>The differences between consecutive elements of an array.</td>
</tr>
<tr>
<td>gradient</td>
<td>Return the gradient of an N-dimensional array.</td>
</tr>
<tr>
<td>cross</td>
<td>Return the cross product of two (arrays of) vectors.</td>
</tr>
<tr>
<td>trapz</td>
<td>Integrate along the given axis using the composite trapezoidal rule.</td>
</tr>
</tbody>
</table>
**prod** (*a*, *axis*=*None*, *dtype*=*None*, *out*=*None*)

Return the product of array elements over a given axis.

**Parameters**

- **a**: array_like
  
  Input data.

- **axis**: int, optional
  
  Axis over which the product is taken. By default, the product of all elements is calculated.

- **dtype**: data-type, optional
  
  The data-type of the returned array, as well as of the accumulator in which the elements are multiplied. By default, if *a* is of integer type, *dtype* is the default platform integer. (Note: if the type of *a* is unsigned, then so is *dtype.*) Otherwise, the dtype is the same as that of *a*.

- **out**: ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

**Returns**

- **product_along_axis**: ndarray, see *dtype* parameter above.
  
  An array shaped as *a* but with the specified axis removed. Returns a reference to *out* if specified.

**See Also:**

- numpy.prod
  
  equivalent method

- numpy.doc.ufuncs
  
  Section “Output arguments”

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```python
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x)  #random
16
```

**Examples**

By default, calculate the product of all elements:

```python
>>> np.prod([1., 2.])
2.0
```

Even when the input array is two-dimensional:

```python
>>> npprod([[1., 2.], [3., 4.]])
24.0
```

But we can also specify the axis over which to multiply:
```python
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([ 2., 12.])
```

If the type of `x` is unsigned, then the output type is the unsigned platform integer:

```python
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
```

If `x` is of a signed integer type, then the output type is the default platform integer:

```python
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == np.int
True
```

**sum (a, axis=None, dtype=None, out=None)**

Sum of array elements over a given axis.

**Parameters**

- **a**: array_like
  Elements to sum.
- **axis**: integer, optional
  Axis over which the sum is taken. By default `axis` is None, and all elements are summed.
- **dtype**: dtype, optional
  The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of `a` is used. An exception is when `a` has an integer type with less precision than the default platform integer. In that case, the default platform integer is used instead.
- **out**: ndarray, optional
  Array into which the output is placed. By default, a new array is created. If `out` is given, it must be of the appropriate shape (the shape of `a` with `axis` removed, i.e., `numpy.delete(a.shape, axis)`). Its type is preserved. See `doc.ufuncs` (Section “Output arguments”) for more details.

**Returns**

- **sum_along_axis**: ndarray
  An array with the same shape as `a`, with the specified axis removed. If `a` is a 0-d array, or if `axis` is None, a scalar is returned. If an output array is specified, a reference to `out` is returned.

**See Also:**

- `ndarray.sum`
  Equivalent method.
- `cumsum`
  Cumulative sum of array elements.
- `trapz`
  Integration of array values using the composite trapezoidal rule.
- `mean, average`
Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```

If the accumulator is too small, overflow occurs:

```python
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

**nansum** *(a, axis=None)*

Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

**Parameters**

- **a**: array_like
  
  Array containing numbers whose sum is desired. If a is not an array, a conversion is attempted.

- **axis**: int, optional
  
  Axis along which the sum is computed. The default is to compute the sum of the flattened array.

**Returns**

- **y**: ndarray
  
  An array with the same shape as a, with the specified axis removed. If a is a 0-d array, or if axis is None, a scalar is returned with the same dtype as a.

**See Also:**

- **numpy.sum**
  Sum across array including Not a Numbers.

- **isnan**
  Shows which elements are Not a Number (NaN).

- **isfinite**
  Shows which elements are not: Not a Number, positive and negative infinity

**Notes**

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infnity. If positive or negative infinity are present the result is positive or negative infinity. But if both positive and negative infinity are present, the result is Not A Number (NaN).

Arithmetic is modular when using integer types (all elements of a must be finite i.e. no elements that are NaNs, positive infinity and negative infinity because NaNs are floating point types), and no error is raised on overflow.
Examples

```python
>>> np.nansum(1)
1
>>> np.nansum([1])
1
>>> np.nansum([1, np.nan])
1.0
>>> a = np.array([[1, 1], [1, np.nan]])
>>> np.nansum(a)
3.0
>>> np.nansum(a, axis=0)
array([ 2., 1.])
```

When positive infinity and negative infinity are present

```python
>>> np.nansum([1, np.nan, np.inf])
inf
>>> np.nansum([1, np.nan, np.NINF])
-inf
>>> np.nansum([1, np.nan, np.inf, np.NINF])
nan
```

cumprod(a, axis=None, dtype=None, out=None)

Return the cumulative product of elements along a given axis.

Parameters

- **a** : array_like
  Input array.
- **axis** : int, optional
  Axis along which the cumulative product is computed. By default the input is flattened.
- **dtype** : dtype, optional
  Type of the returned array, as well as of the accumulator in which the elements are multiplied. If dtype is not specified, it defaults to the dtype of `a`, unless `a` has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.
- **out** : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

Returns

- **cumprod** : ndarray
  A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

See Also:

numpy.doc.ufuncs

Section “Output arguments”
Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> a = np.array([1, 2, 3])
>>> np.cumprod(a) # intermediate results 1, 1*2
... # total product 1*2*3 = 6
array([1, 2, 6])

>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.cumprod(a, dtype=float) # specify type of output
array([ 1.,  2.,  6., 24., 120., 720.])
```

The cumulative product for each column (i.e., over the rows of) `a`:

```python
>>> np.cumprod(a, axis=0)
array([[ 1.,  2.,  3.],
       [ 4., 10., 18.]])
```

The cumulative product for each row (i.e. over the columns of) `a`:

```python
>>> np.cumprod(a, axis=1)
array([[ 1.,  2.,  6.],
       [ 4., 20., 120.]])
```

cumsum(a, axis=None, dtype=None, out=None)

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

Parameters

- **a**: array_like
  - Input array.
- **axis**: int, optional
  - Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- **dtype**: dtype, optional
  - Type of the returned array and of the accumulator in which the elements are summed. If `dtype` is not specified, it defaults to the dtype of `a`, unless `a` has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.
- **out**: ndarray, optional
  - Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See `doc.ufuncs` (Section “Output arguments”) for more details.

Returns

- **cumsum_along_axis**: ndarray.
  - A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned. The result has the same size as `a`, and the same shape as `a` if `axis` is not None or `a` is a 1-d array.

See Also:
**sum**

Sum array elements.

**trapz**

Integration of array values using the composite trapezoidal rule.

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
goldenrule
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.cumsum(a)
array([1, 3, 6, 10, 15, 21])
>>> np.cumsum(a, dtype=float)  # specifies type of output value(s)
array([1., 3., 6., 10., 15., 21.])
>>> np.cumsum(a, axis=0)  # sum over rows for each of the 3 columns
array([[1, 3, 6],
       [4, 7, 9]])
>>> np.cumsum(a, axis=1)  # sum over columns for each of the 2 rows
array([[1, 3, 6],
       [4, 9, 15]])
```

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

Calculate the n-th order discrete difference along given axis.

The first order difference is given by \(a[n+1] - a[n]\) along the given axis, higher order differences are calculated by using **diff** recursively.

**Parameters**

- **a** : array_like
  - Input array

- **n** : int, optional
  - The number of times values are differenced.

- **axis** : int, optional
  - The axis along which the difference is taken, default is the last axis.

**Returns**

- **out** : ndarray
  - The \(n\) order differences. The shape of the output is the same as \(a\) except along \(axis\) where the dimension is smaller by \(n\).

**See Also:**

- gradient, ediff1d

**Examples**

```python
goldenrule
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.diff(x)
```
array([ 1,  2,  3, -7])

>>> np.diff(x, n=2)
array([ 1,  1, -10])

>>> x = np.array([[1, 3, 6, 10], [0, 5, 6, 8]])

>>> np.diff(x)
array([[2, 3, 4],
       [5, 1, 2]])

>>> np.diff(x, axis=0)
array([[-1, 2, 0, -2]])

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

The differences between consecutive elements of an array.

Parameters

ary : array_like
    If necessary, will be flattened before the differences are taken.

to_end : array_like, optional
    Number(s) to append at the end of the returned differences.

to_begin : array_like, optional
    Number(s) to prepend at the beginning of the returned differences.

Returns

ed : ndarray
    The differences. Loosely, this is ary.flat[1:] - ary.flat[:-1].

See Also:
diff, gradient

Notes

When applied to masked arrays, this function drops the mask information if the to_begin and/or to_end parameters are used.

Examples

>>> x = np.array([1, 2, 4, 7, 0])

>>> np.ediff1d(x)
array([ 1,  2,  3, -7])

>>> np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
array([-99,  1,  2,  3, -7,  88,  99])

The returned array is always 1D.

>>> y = [[1, 2, 4], [1, 6, 24]]

>>> np.ediff1d(y)
array([ 1, -3,  5, 18])

gradient(f, *varargs)

Return the gradient of an N-dimensional array.

3.13. Mathematical functions
The gradient is computed using central differences in the interior and first differences at the boundaries. The returned gradient hence has the same shape as the input array.

**Parameters**

- **f**: array_like
  
  An N-dimensional array containing samples of a scalar function.

- **varargs**: scalars
  
  0, 1, or N scalars specifying the sample distances in each direction, that is: \(dx, dy, dz,\) ...
  
  The default distance is 1.

**Returns**

- **g**: ndarray
  
  N arrays of the same shape as \(f\) giving the derivative of \(f\) with respect to each dimension.

**Examples**

```python
t> x = np.array([1, 2, 4, 7, 11, 16], dtype=np.float)
t> np.gradient(x)
anarray([ 1. , 1.5, 2.5, 3.5, 4.5, 5. ])
t> np.gradient(x, 2)
anarray([ 0.5 , 0.75, 1.25, 1.75, 2.25, 2.5 ])
```

**cross**

```python
(a, b, axisa=-1, axesb=-1, axisc=-1, axis=None)

Return the cross product of two (arrays of) vectors.
```

The cross product of \(a\) and \(b\) in \(R^3\) is a vector perpendicular to both \(a\) and \(b\). If \(a\) and \(b\) are arrays of vectors, the vectors are defined by the last axis of \(a\) and \(b\) by default, and these axes can have dimensions 2 or 3. Where the dimension of either \(a\) or \(b\) is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

**Parameters**

- **a**: array_like
  
  Components of the first vector(s).

- **b**: array_like
  
  Components of the second vector(s).

- **axisa**: int, optional
  
  Axis of \(a\) that defines the vector(s). By default, the last axis.

- **axesb**: int, optional
  
  Axis of \(b\) that defines the vector(s). By default, the last axis.

- **axisc**: int, optional
  
  Axis of \(c\) containing the cross product vector(s). By default, the last axis.

- **axis**: int, optional
If defined, the axis of $a$, $b$ and $c$ that defines the vector(s) and cross product(s). Overrides $axisa$, $axisb$ and $axisc$.

**Returns**
- $c$ : ndarray
  Vector cross product(s).

**Raises**
- `ValueError`
  When the dimension of the vector(s) in $a$ and/or $b$ does not equal 2 or 3.

**See Also:**
- `inner`
  Inner product
- `outer`
  Outer product.
- `ix_`
  Construct index arrays.

**Examples**

Vector cross-product.

```python
def x = [1, 2, 3]
def y = [4, 5, 6]
def np.cross(x, y)
array([-3, 6, -3])
```

One vector with dimension 2.

```python
def x = [1, 2]
def y = [4, 5, 6]
def np.cross(x, y)
array([12, -6, -3])
```

Equivalently:

```python
def x = [1, 2, 0]
def y = [4, 5, 6]
def np.cross(x, y)
array([12, -6, -3])
```

Both vectors with dimension 2.

```python
def x = [1, 2]
def y = [4, 5]
def np.cross(x, y)
-3
```

Multiple vector cross-products. Note that the direction of the cross product vector is defined by the *right-hand rule*.
```python
>>> x = np.array([[1,2,3], [4,5,6]])
>>> y = np.array([[4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[ 3, -6,  3],
       [ 3, -6,  3]])
```

The orientation of \( c \) can be changed using the \textit{axis} \textit{c} keyword.

```python
>>> np.cross(x, y, axis_c=0)
array([[-3,  6, -3],
       [ 3, -6,  3]])
```

Change the vector definition of \( x \) and \( y \) using \textit{axis} a and \textit{axis} b.

```python
>>> x = np.array([[1,2,3], [4,5,6], [7, 8, 9]])
>>> y = np.array([[7, 8, 9], [4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[-6, 12, -6],
       [ 0,  0,  0],
       [ 6, -12,  6]])
>>> np.cross(x, y, axisa=0, axisb=0)
array([[-24, 48, -24],
       [ 30, -60, 30],
       [ 36, -72, 36]])
```

\texttt{trapz}(y, x=None, dx=1.0, axis=-1)

Integrate along the given axis using the composite trapezoidal rule.

Integrate \( y(x) \) along given axis.

Parameters

\( y \) : array_like

Input array to integrate.

\( x \) : array_like, optional

If \( x \) is None, then spacing between all \( y \) elements is \( dx \).

\( dx \) : scalar, optional

If \( x \) is None, spacing given by \( dx \) is assumed. Default is 1.

\( axis \) : int, optional

Specify the axis.

Returns

\( out \) : float

Definite integral as approximated by trapezoidal rule.

See Also:

\texttt{sum}, \texttt{cumsum}

Notes

Image [R143] illustrates trapezoidal rule – y-axis locations of points will be taken from \( y \) array, by default x-axis distances between points will be 1.0, alternatively they can be provided with \( x \) array or with \( dx \) scalar. Return value will be equal to combined area under the red lines.
3.13.5 Exponents and logarithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exp (x) )</td>
<td>Calculate the exponential of all elements in the input array.</td>
</tr>
<tr>
<td>( \expm1 (x) )</td>
<td>Calculate ( \exp (x) - 1 ) for all elements in the array.</td>
</tr>
<tr>
<td>( \exp2 (x) )</td>
<td>Calculate ( 2^x ) for all elements in the input array.</td>
</tr>
<tr>
<td>( \log (x) )</td>
<td>Natural logarithm, element-wise.</td>
</tr>
<tr>
<td>( \log10 (x) )</td>
<td>Return the base 10 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td>( \log2 (x, y) )</td>
<td>Return the base 2 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td>( \log1p (x) )</td>
<td>Return the natural logarithm of one plus the input array, element-wise.</td>
</tr>
<tr>
<td>( \logaddexp (x, y) )</td>
<td>Logarithm of the sum of exponentiations of the inputs.</td>
</tr>
<tr>
<td>( \logaddexp2 (x, y) )</td>
<td>Logarithm of the sum of exponentiations of the inputs in base-2.</td>
</tr>
</tbody>
</table>

**exp**

Calculate the exponential of all elements in the input array.

**Parameters**

- **x** : array_like
  - Input values.

**Returns**

- **out** : ndarray
  - Output array, element-wise exponential of \( x \).

**See Also:**

- \( \expm1 \)
  - Calculate \( \exp (x) - 1 \) for all elements in the array.

- \( \exp2 \)
  - Calculate \( 2**x \) for all elements in the array.

**Notes**

The irrational number \( e \) is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, \( \ln \) (this means that, if \( x = \ln y = \log_e y \), then \( e^x = y \). For real input, \( \exp (x) \) is always...
positive.

For complex arguments, \( x = a + ib \), we can write \( e^x = e^a e^{ib} \). The first term, \( e^a \), is already known (it is the real argument, described above). The second term, \( e^{ib} \), is \( \cos b + i \sin b \), a function with magnitude 1 and a periodic phase.

**References**

[R28], [R29]

**Examples**

Plot the magnitude and phase of \( \exp(x) \) in the complex plane:

```python
>>> import matplotlib.pyplot as plt

>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis]  # a + ib over complex plane
>>> out = np.exp(xx)

>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
... extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Magnitude of exp(x)')

>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
... extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

**expm1()**

Calculate \( \exp(x) - 1 \) for all elements in the array.

**Parameters**

\( x \) : array_like

Input values.
Returns

\[ \text{out} : \text{ndarray} \]

Element-wise exponential minus one: \[ \text{out} = \exp(x) - 1 \].

See Also:

\textbf{log1p}

\[ \log(1 + x) \], the inverse of \text{expm1}.\]

Notes

This function provides greater precision than the formula \( \exp(x) - 1 \) for small values of \( x \).

Examples

The true value of \( \exp(1e-10) - 1 \) is 1.00000000005e-10 to about 32 significant digits. This example shows the superiority of \text{expm1} in this case.

```python
>>> np.expm1(1e-10)
1.00000000005e-10
```

\textbf{exp2 ()}

Calculate \( 2^p \) for all \( p \) in the input array.

Parameters

\[ x : \text{array-like} \]

Input values.

\[ \text{out} : \text{ndarray}, \text{optional} \]

Array to insert results into.

Returns

\[ \text{out} : \text{ndarray} \]

Element-wise 2 to the power \( x \).

See Also:

\textbf{exp}

\[ \text{calculate } x^p. \]

Notes

New in version 1.3.0.

Examples

```python
>>> np.exp2([2, 3])
array([ 4., 8.])
```

\textbf{log ()}

Natural logarithm, element-wise.

The natural logarithm \( \log \) is the inverse of the exponential function, so that \( \log(\exp(x)) = x \). The natural logarithm is logarithm in base \( e \).
NumPy Reference, Release 1.4.1

Parameters
   x : array_like
       Input value.

Returns
   y : ndarray
       The natural logarithm of x, element-wise.

See Also:
   log10, log2, log1p, emath.log

Notes
Logarithm is a multivalued function: for each x there is an infinite number of z such that exp(z) = x. The convention is to return the z whose imaginary part lies in [-pi, pi].

For real-valued input data types, log always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, log is a complex analytical function that has a branch cut [-inf, 0] and is continuous from above on it. log handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References
   [R51], [R52]

Examples
   >>> np.log([1, np.e, np.e**2, 0])
   array([ 0., 1., 2., -Inf])

log10()
Return the base 10 logarithm of the input array, element-wise.

Parameters
   x : array_like
       Input values.

Returns
   y : ndarray
       The logarithm to the base 10 of x, element-wise. NaNs are returned where x is negative.

See Also:
   emath.log10

Notes
Logarithm is a multivalued function: for each x there is an infinite number of z such that 10**z = x. The convention is to return the z whose imaginary part lies in [-pi, pi].

For real-valued input data types, log10 always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, log10 is a complex analytical function that has a branch cut [-inf, 0] and is continuous from above on it. log10 handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.
**References**

[R53], [R54]

**Examples**

```python
>>> np.log10([1e-15, -3.])
array([-15., NaN])
```

**log2** *(x, y=None)*  
Return the base 2 logarithm of the input array, element-wise.

**Parameters**

- **x** : array_like  
  Input array.

- **y** : array_like  
  Optional output array with the same shape as `x`.

**Returns**

- **y** : ndarray  
  The logarithm to the base 2 of `x` element-wise. NaNs are returned where `x` is negative.

**See Also:**

`log`, `log1p`, `log10`

**Examples**

```python
>>> np.log2([-1, 2, 4])
array([ NaN,  1.,  2.])
```

**log1p()**  
Return the natural logarithm of one plus the input array, element-wise.

Calculates \( \log(1 + x) \).

**Parameters**

- **x** : array_like  
  Input values.

**Returns**

- **y** : ndarray  
  Natural logarithm of \( 1 + x \), element-wise.

**See Also:**

`expm1`

\( \exp(x) - 1 \), the inverse of `log1p`.

**Notes**

For real-valued input, `log1p` is accurate also for `x` so small that \( 1 + x == 1 \) in floating-point accuracy.

Logarithm is a multivalued function: for each \( x \) there is an infinite number of \( z \) such that \( \exp(z) = 1 + x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi, \pi]\).
For real-valued input data types, \texttt{log1p} always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \texttt{nan} and sets the \texttt{invalid} floating point error flag.

For complex-valued input, \texttt{log1p} is a complex analytical function that has a branch cut \([-\infty, -1]\) and is continuous from above on it. \texttt{log1p} handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

**References**

[R55], [R56]

**Examples**

```python
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

\textbf{logaddexp} ()

Logarithm of the sum of exponentiations of the inputs.

Calculates \(\log(\exp(x1) + \exp(x2))\). This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

**Parameters**

\(x1, x2\) : array_like

Input values.

**Returns**

\texttt{result} : ndarray

Logarithm of \(\exp(x1) + \exp(x2)\).

**See Also:**

\textbf{logaddexp2}

Logarithm of the sum of exponentiations of inputs in base-2.

**Notes**

New in version 1.3.0.

**Examples**

```python
>>> prob1 = np.log(1e-50)
>>> prob2 = np.log(2.5e-50)
>>> prob12 = np.logaddexp(prob1, prob2)
>>> prob12
-113.87649168120691
>>> np.exp(prob12)
3.500000000000057e-50
```

\textbf{logaddexp2} ()

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates \(\log2(2**x1 + 2**x2)\). This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the
base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

**Parameters**

- `x1, x2 : array_like`
  Input values.
- `out : ndarray, optional`
  Array to store results in.

**Returns**

- `result : ndarray`
  Base-2 logarithm of $2^{x1} + 2^{x2}$.

**See Also:**

- `logaddexp`
  Logarithm of the sum of exponentiations of the inputs.

**Notes**

New in version 1.3.0.

**Examples**

```python
>>> prob1 = np.log2(1e-50)
>>> prob2 = np.log2(2.5e-50)
>>> prob12 = np.logaddexp2(prob1, prob2)
>>> prob1, prob2, prob12
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> 2**prob12
3.4999999999999914e-50
```

### 3.13.6 Other special functions

#### `i0 (x)`
Modified Bessel function of the first kind, order 0.

Return the sinc function.

#### `sinc (x)`
Modified Bessel function of the first kind, order 0.

Usually denoted $I_0$. This function does broadcast, but will *not* “up-cast” int dtype arguments unless accompanied by at least one float or complex dtype argument (see Raises below).

**Parameters**

- `x : array_like, dtype float or complex`
  Argument of the Bessel function.

**Returns**

- `out : ndarray, shape = x.shape, dtype = x.dtype`
  The modified Bessel function evaluated at each of the elements of `x`.

**Raises**

- `TypeError: array cannot be safely cast to required type`
  If argument consists exclusively of int dtypes.
See Also:

`scipy.special.iv`, `scipy.special.ive`

Notes

We use the algorithm published by Clenshaw [R38] and referenced by Abramowitz and Stegun [R39], for which
the function domain is partitioned into the two intervals [0,8] and (8,inf), and Chebyshev polynomial expansions
are employed in each interval. Relative error on the domain [0,30] using IEEE arithmetic is documented [R40]
as having a peak of 5.8e-16 with an rms of 1.4e-16 (n = 30000).

References

[R38], [R39], [R40]

Examples

```python
>>> np.i0([0.])
array(1.0)
>>> np.i0([0., 1. + 2j])
array([ 1.00000000+0.j , 0.18785373+0.64616944j])
```

`sinc(x)`

Return the sinc function.

The sinc function is $\frac{\sin(\pi x)}{\pi x}$.

Parameters

- `x` : ndarray
  Array (possibly multi-dimensional) of values for which to calculate sinc(x).

Returns

- `out` : ndarray
  sinc(x), which has the same shape as the input.

Notes

sinc(0) is the limit value 1.

The name sinc is short for “sine cardinal” or “sinus cardinalis”.

The sinc function is used in various signal processing applications, including in anti-aliasing, in the construction
of a Lanczos resampling filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal interpolation kernel is proportional to the sinc function.

References

[R138], [R139]

Examples

```python
>>> x = np.arange(-20., 21.)/5.
```

```python
>>> np.sinc(x)
array([-3.89804309e-17, -4.92362781e-02, -8.40918587e-02, -8.90384387e-02,
      -5.84680802e-02, -3.89804309e-02,  1.6434881e-01,  1.26137788e-01,
      0.18785373+0.64616944j,  0.18785373+0.64616944j])
```
NumPy Reference, Release 1.4.1

>>> import matplotlib.pyplot as plt
>>> plt.plot(x, np.sinc(x))
>>> plt.title("Sinc Function")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("X")
>>> plt.show()

It works in 2-D as well:

>>> x = np.arange(-200., 201.)/50.
>>> xx = np.outer(x, x)
>>> plt.imshow(np.sinc(xx))

3.13.7 Floating point routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{signbit (x[, out])}</td>
<td>Returns element-wise True where signbit is set (less than zero).</td>
</tr>
<tr>
<td>\texttt{copysign (x1, x2[, out])}</td>
<td>Change the sign of \texttt{x1} to that of \texttt{x2}, element-wise.</td>
</tr>
<tr>
<td>\texttt{frexp (x[, out1, out2])}</td>
<td>Split the number, \texttt{x}, into a normalized fraction (\texttt{y1}) and exponent (\texttt{y2})</td>
</tr>
<tr>
<td>\texttt{ldexp (x1, x2[, out])}</td>
<td>Compute \texttt{y = x1 * 2**x2}.</td>
</tr>
</tbody>
</table>

\texttt{signbit (x)}

Returns element-wise True where signbit is set (less than zero).

Parameters

\texttt{x: array_like :}
The input value(s).

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

out : array_like, bool

Output array.

Examples

>>> np.signbit(-1.2)
True

>>> np.signbit(np.array([1, -2.3, 2.1]))
array([False, True, False], dtype=bool)

copysign()

Change the sign of x1 to that of x2, element-wise.

If both arguments are arrays or sequences, they have to be of the same length. If x2 is a scalar, its sign will be copied to all elements of x1.

Parameters

x1: array_like :

Values to change the sign of.

x2: array_like :

The sign of x2 is copied to x1.

out : ndarray, optional

Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

out : array_like

The values of x1 with the sign of x2.

Examples

>>> np.copysign(1.3, -1)
-1.3

>>> 1/np.copysign(0, 1)
inf

>>> 1/np.copysign(0, -1)
-inf

>>> np.copysign([-1, 0, 1], -1.1)
array([-1., -0., -1.])

>>> np.copysign([-1, 0, 1], np.arange(3)-1)
array([-1., 0., 1.])

frexp()

Split the number, x, into a normalized fraction (y1) and exponent (y2)
`ldexp()`

Compute \( y = x1 * 2^{x2} \).

### 3.13.8 Arithmetic operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x1, x2[, out])</code></td>
<td>Add arguments element-wise.</td>
</tr>
<tr>
<td><code>reciprocal(x[, out])</code></td>
<td>Return the reciprocal of the argument, element-wise.</td>
</tr>
<tr>
<td><code>negative(x[, out])</code></td>
<td>Returns an array with the negative of each element of the original array.</td>
</tr>
<tr>
<td><code>multiply(x1, x2[, out])</code></td>
<td>Multiply arguments element-wise.</td>
</tr>
<tr>
<td><code>divide(x1, x2[, out])</code></td>
<td>Divide arguments element-wise.</td>
</tr>
<tr>
<td><code>power(x1, x2[, out])</code></td>
<td>Returns element-wise base array raised to power from second array.</td>
</tr>
<tr>
<td><code>subtract(x1, x2[, out])</code></td>
<td>Subtract arguments, element-wise.</td>
</tr>
<tr>
<td><code>true_divide(x1, x2[, out])</code></td>
<td>Returns a true division of the inputs, element-wise.</td>
</tr>
<tr>
<td><code>floor_divide(x1, x2[, out])</code></td>
<td>Return the largest integer smaller or equal to the division of the inputs.</td>
</tr>
<tr>
<td><code>fmod(x1, x2[, out])</code></td>
<td>Return the element-wise remainder of division.</td>
</tr>
<tr>
<td><code>mod(x1, x2[, out])</code></td>
<td>Return element-wise remainder of division.</td>
</tr>
<tr>
<td><code>modf(x[, out1, out2])</code></td>
<td>Return the fractional and integral parts of an array, element-wise.</td>
</tr>
<tr>
<td><code>remainder(x1, x2[, out])</code></td>
<td>Return element-wise remainder of division.</td>
</tr>
</tbody>
</table>

#### `add()`

Add arguments element-wise.

**Parameters**

- `x1`, `x2` : array_like
  
The arrays to be added.

**Returns**

- `y` : {ndarray, scalar}
  
The sum of `x1` and `x2`, element-wise. Returns scalar if both `x1` and `x2` are scalars.

**Notes**

Equivalent to `x1 + x2` in terms of array broadcasting.

**Examples**

```python
>>> np.add(1.0, 4.0)
5.0
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.add(x1, x2)
array([[ 0.,  2.,  4.],
       [ 3.,  5.,  7.],
       [ 6.,  8., 10.]])
```

#### `reciprocal()`

Return the reciprocal of the argument, element-wise.

Calculates \( 1 / x \).

**Parameters**

- `x` : array_like
  
  Input array.

**Returns**

- `y` : ndarray
  
  The reciprocal of the argument, element-wise.
Return array.

Notes

Note: This function is not designed to work with integers.

For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

Examples

```python
>>> np.reciprocal(2.)
0.5
>>> np.reciprocal([1, 2., 3.33])
array([ 1.     ,  0.5    ,  0.3003003])
```

negative()

Returns an array with the negative of each element of the original array.

Parameters

- x: {array_like, scalar}
  Input array.

Returns

- y: {ndarray, scalar}
  Returned array or scalar $y=-x$.

Examples

```python
>>> np.negative([1.,-1.])
array([-1., 1.])
```

multiply()

Multiply arguments element-wise.

Parameters

- x1, x2: array_like
  Input arrays to be multiplied.

Returns

- y: ndarray
  The product of $x1$ and $x2$, element-wise. Returns a scalar if both $x1$ and $x2$ are scalars.

Notes

Equivalent to $x1 \times x2$ in terms of array broadcasting.

Examples

```python
>>> np.multiply(2.0, 4.0)
8.0
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.multiply(x1, x2)
array([[ 0.,  1.,  4.],
       [ 2.,  4.,  8.],
       [ 4.,  8., 16.]])
```
divide()
Divide arguments element-wise.

Parameters
- x1 : array_like
  Dividend array.
- x2 : array_like
  Divisor array.
- out : ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns
- y : {ndarray, scalar}
  The quotient x1/x2, element-wise. Returns a scalar if both x1 and x2 are scalars.

See Also:
- seterr
  Set whether to raise or warn on overflow, underflow and division by zero.

Notes
Equivalent to x1 / x2 in terms of array-broadcasting.

Behavior on division by zero can be changed using seterr.

When both x1 and x2 are of an integer type, divide will return integers and throw away the fractional part. Moreover, division by zero always yields zero in integer arithmetic.

Examples

```python
>>> np.divide(2.0, 4.0)
0.5
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.divide(x1, x2)
array([[ NaN,  1.,  1.],
       [ Inf,  4.,  2.5],
       [ Inf,  7.,  4. ]])
```

Note the behavior with integer types:

```python
>>> np.divide(2, 4)
0
>>> np.divide(2, 4.)
0.5
```

Division by zero always yields zero in integer arithmetic, and does not raise an exception or a warning:

```python
>>> np.divide(np.array([0, 1], dtype=int), np.array([0, 0], dtype=int))
array([0, 0])
```
Division by zero can, however, be caught using \texttt{seterr}:

```python
>>> old_err_state = np.seterr(divide='raise')
>>> np.divide(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: divide by zero encountered in divide

>>> ignored_states = np.seterr(**old_err_state)
>>> np.divide(1, 0)
0
```

\texttt{power}()

Returns element-wise base array raised to power from second array.

Raise each base in \texttt{x1} to the power of the exponents in \texttt{x2}. This requires that \texttt{x1} and \texttt{x2} must be broadcastable to the same shape.

**Parameters**

- \texttt{x1}: array_like
  - The bases.
- \texttt{x2}: array_like
  - The exponents.

**Returns**

\texttt{y}: ndarray

The bases in \texttt{x1} raised to the exponents in \texttt{x2}.

**Examples**

Cube each element in a list.

```python
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> np.power(x1, 3)
array([ 0,  1,  8, 27, 64, 125])
```

Raise the bases to different exponents.

```python
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> np.power(x1, x2)
array([ 0.,  1.,  8., 27., 16.,  5.])
```

The effect of broadcasting.

```python
>>> x2 = np.array([[[1, 2, 3, 3, 2], [1, 2, 3, 3, 2]]])
>>> x2
array([[[1, 2, 3, 3, 2],
         [1, 2, 3, 3, 2]]])
>>> np.power(x1, x2)
array([[[ 0,  1,  8, 27, 16,  5],
         [ 0,  1,  8, 27, 16,  5]]])
```

\texttt{subtract}()

Subtract arguments, element-wise.
Parameters
   x1, x2 : array_like
   The arrays to be subtracted from each other.

Returns
   y : ndarray
   The difference of x1 and x2, element-wise. Returns a scalar if both x1 and x2 are scalars.

Notes
   Equivalent to x1 - x2 in terms of array broadcasting.

Examples
   >>> np.subtract(1.0, 4.0)
   -3.0
   >>> x1 = np.arange(9.0).reshape((3, 3))
   >>> x2 = np.arange(3.0)
   >>> np.subtract(x1, x2)
   array([[ 0.,  0.,  0.],
          [ 3.,  3.,  3.],
          [ 6.,  6.,  6.]])

true_divide()
   Returns a true division of the inputs, element-wise.
   Instead of the Python traditional ‘floor division’, this returns a true division. True division adjusts the output type to present the best answer, regardless of input types.

Parameters
   x1 : array_like
       Dividend array.
   x2 : array_like
       Divisor array.

Returns
   out : ndarray
       Result is scalar if both inputs are scalar, ndarray otherwise.

Notes
   The floor division operator // was added in Python 2.2 making // and / equivalent operators. The default floor division operation of / can be replaced by true division with from __future__ import division.
   In Python 3.0, // is the floor division operator and / the true division operator. The true_divide(x1, x2) function is equivalent to true division in Python.

Examples
   >>> x = np.arange(5)
   >>> np.true_divide(x, 4)
   array([ 0. ,  0.25,  0.5 ,  0.75,  1. ])

3.13. Mathematical functions
>>> x/4
array([0, 0, 0, 0, 1])
>>> x//4
array([0, 0, 0, 0, 1])

>>> from __future__ import division
>>> x/4
array([ 0. , 0.25, 0.5 , 0.75, 1. ])
>>> x//4
array([0, 0, 0, 0, 1])

floor_divide()
Return the largest integer smaller or equal to the division of the inputs.

Parameters
x1 : array_like
   Numerator.

x2 : array_like
   Denominator.

Returns
y : ndarray
   y = floor(x1/x2)

See Also:
divide
   Standard division.

floor
   Round a number to the nearest integer toward minus infinity.

ceil
   Round a number to the nearest integer toward infinity.

Examples

>>> np.floor_divide(7,3)
2
>>> np.floor_divide([1., 2., 3., 4.], 2.5)
array([ 0., 0., 1., 1.])

fmod()  
Return the element-wise remainder of division.
This is the NumPy implementation of the Python modulo operator %.

Parameters

x1 : array_like
   Dividend.

x2 : array_like
   Divisor.
Returns
   y : array_like
       The remainder of the division of x1 by x2.

See Also:

remainder
   Modulo operation where the quotient is floor(x1/x2).

divide

Notes
The result of the modulo operation for negative dividend and divisors is bound by conventions. In fmod, the sign of the remainder is the sign of the dividend. In remainder, the sign of the divisor does not affect the sign of the result.

Examples

```python
>>> np.fmod([-3, -2, -1, 1, 2, 3], 2)
array([-1, 0, -1, 1, 0, 1])
>>> np.remainder([-3, -2, -1, 1, 2, 3], 2)
array([1, 0, 1, 1, 0, 1])
>>> np.fmod([5, 3], [2, 2.])
array([ 1., 1.])
```

mod()
   Return element-wise remainder of division.

Computes x1 - floor(x1 / x2) * x2.

Parameters
   x1 : array_like
       Dividend array.

   x2 : array_like
       Divisor array.

   out : ndarray, optional
       Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns
   y : ndarray
       The remainder of the quotient x1/x2, element-wise. Returns a scalar if both x1 and x2 are scalars.
See Also:
divide, floor

Notes
Returns 0 when \(x_2\) is 0 and both \(x_1\) and \(x_2\) are (arrays of) integers.

Examples

```python
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

**modf()**
Return the fractional and integral parts of an array, element-wise.
The fractional and integral parts are negative if the given number is negative.

Parameters
- **x**: array_like
  Input array.

Returns
- **y1**: ndarray
  Fractional part of \(x\).
- **y2**: ndarray
  Integral part of \(x\).

Notes
For integer input the return values are floats.

Examples

```python
>>> np.modf([0, 3.5])
(array([ 0. , 0.5]), array([ 0., 3.]))
>>> np.modf(-0.5)
(-0.5, -0)
```

**remainder()**
Return element-wise remainder of division.
Computes \(x_1 - \text{floor}(x_1 / x_2) \times x_2\).

Parameters
- **x1**: array_like
  Dividend array.
- **x2**: array_like
  Divisor array.
- **out**: ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.
Returns

\[ y : \text{ndarray} \]

The remainder of the quotient \( x_1 / x_2 \), element-wise. Returns a scalar if both \( x_1 \) and \( x_2 \) are scalars.

See Also:

divide, floor

Notes

Returns 0 when \( x_2 \) is 0 and both \( x_1 \) and \( x_2 \) are (arrays of) integers.

Examples

```python
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

3.13.9 Handling complex numbers

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>Return the angle of the complex argument.</td>
</tr>
<tr>
<td>real</td>
<td>Return the real part of the elements of the array.</td>
</tr>
<tr>
<td>imag</td>
<td>Return the imaginary part of the elements of the array.</td>
</tr>
<tr>
<td>conj</td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
</tbody>
</table>

angle\((z, \text{deg}=0)\)

Return the angle of the complex argument.

Parameters

- \( z \) : array_like

  A complex number or sequence of complex numbers.

- \( \text{deg} \) : bool, optional

  Return angle in degrees if True, radians if False (default).

Returns

angle : {ndarray, scalar}

The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

See Also:

arctan2, absolute

Examples

```python
>>> np.angle([1.0, 1.0j, 1+1j]) # in radians
array([ 0.        , 1.57079633, 0.78539816])
>>> np.angle(1+1j, deg=True)  # in degrees
45.0
```

real\((\text{val})\)

Return the real part of the elements of the array.
Parameters
   val : array_like
         Input array.

Returns
   out : ndarray
         Output array. If val is real, the type of val is used for the output. If val has complex elements, the returned type is float.

See Also:
   real_if_close, imag, angle

Examples

>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.real
array([ 1.,  3.,  5.])
>>> a.real = 9
>>> a
array([ 9.+2.j,  9.+4.j,  9.+6.j])
>>> a.real = np.array([9, 8, 7])
>>> a
array([ 9.+2.j,  8.+4.j,  7.+6.j])

imag(val)
Return the imaginary part of the elements of the array.

Parameters
   val : array_like
         Input array.

Returns
   out : ndarray
         Output array. If val is real, the type of val is used for the output. If val has complex elements, the returned type is float.

See Also:
   real, angle, real_if_close

Examples

>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.imag
array([ 2.,  4.,  6.])
>>> a.imag = np.array([8, 10, 12])
>>> a
array([ 1.+8.j,  3.+10.j,  5.+12.j])

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

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters
   x : array_like
Input value.

**Returns**

- **y**: ndarray

  The complex conjugate of x, with same dtype as y.

**Examples**

```python
def np.conjugate(a):
    return np.conjugate(a)
```

```python
def np.conjugate(x):
    return np.conjugate(x)
```

```python
>>> np.conjugate(1+2j)
(1-2j)
```

```python
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-0.j, 0.-0.j],
       [ 0.-0.j, 1.-0.j]])
```

### 3.13.10 Miscellaneous

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convolve(a, v[, mode])</code></td>
<td>Returns the discrete, linear convolution of two one-dimensional sequences.</td>
</tr>
<tr>
<td><code>clip(a, a_min, a_max[, out])</code></td>
<td>Clip (limit) the values in an array.</td>
</tr>
<tr>
<td><code>sqrt(x[, out])</code></td>
<td>Return the positive square-root of an array, element-wise.</td>
</tr>
<tr>
<td><code>square(x[, out])</code></td>
<td>Return the element-wise square of the input.</td>
</tr>
<tr>
<td><code>fabs(x[, out])</code></td>
<td>Calculate the absolute value element-wise.</td>
</tr>
<tr>
<td><code>sign(x[, out])</code></td>
<td>Compute the absolute values elementwise.</td>
</tr>
<tr>
<td><code>maximum(x1, x2[, out])</code></td>
<td>Returns an element-wise indication of the sign of a number.</td>
</tr>
<tr>
<td><code>minimum(x1, x2[, out])</code></td>
<td>Element-wise maximum of array elements.</td>
</tr>
<tr>
<td><code>nan_to_num(x)</code></td>
<td>Element-wise minimum of array elements.</td>
</tr>
<tr>
<td><code>real_if_close(a[, tol])</code></td>
<td>Replace nan with zero and inf with finite numbers.</td>
</tr>
<tr>
<td><code>interp(x, xp, fp[, left, right])</code></td>
<td>If complex input returns a real array if complex parts are close to zero.</td>
</tr>
<tr>
<td><code>interpolate(x, xp, fp[, left, right])</code></td>
<td>One-dimensional linear interpolation.</td>
</tr>
</tbody>
</table>

**convolve**

Returns the discrete, linear convolution of two one-dimensional sequences.

The convolution operator is often seen in signal processing, where it models the effect of a linear time-invariant system on a signal [R27]. In probability theory, the sum of two independent random variables is distributed according to the convolution of their individual distributions.

**Parameters**

- **a**: (N,) array_like
  - First one-dimensional input array.
- **v**: (M,) array_like
  - Second one-dimensional input array.
- **mode**: {'full', 'valid', 'same'}, optional
  - **‘full’**: By default, mode is ‘full’. This returns the convolution at each point of overlap, with an output shape of (N+M-1). At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen.
  - **‘same’**: Mode same returns output of length max(M, N). Boundary effects are still visible.
Mode `valid` returns output of length \( \max(M, N) - \min(M, N) + 1 \). The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

Returns

out : ndarray
Discrete, linear convolution of \( a \) and \( v \).

See Also:

- `scipy.signal.fftconv`  
  Convolve two arrays using the Fast Fourier Transform.
- `scipy.linalg.toeplitz`  
  Used to construct the convolution operator.

Notes

The discrete convolution operation is defined as

\[
(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n-m]
\]

It can be shown that a convolution \( x(t) * y(t) \) in time/space is equivalent to the multiplication \( X(f)Y(f) \) in the Fourier domain, after appropriate padding (padding is necessary to prevent circular convolution). Since multiplication is more efficient (faster) than convolution, the function `scipy.signal.fftconvolve` exploits the FFT to calculate the convolution of large data-sets.

References

[R27]

Examples

Note how the convolution operator flips the second array before “sliding” the two across one another:

```python
>>> np.convolve([1, 2, 3], [0, 1, 0.5])
array([ 0. , 1. , 2.5, 4. , 1.5])
```

Only return the middle values of the convolution. Contains boundary effects, where zeros are taken into account:

```python
>>> np.convolve([1,2,3],[0,1,0.5], 'same')
array([ 1. , 2.5, 4. ])
```

The two arrays are of the same length, so there is only one position where they completely overlap:

```python
>>> np.convolve([1,2,3],[0,1,0.5], 'valid')
array([ 2.5])
```

`clip(a, a_min, a_max, out=None)`  
Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of \([0, 1]\) is specified, values smaller than 0 become 0, and values larger than 1 become 1.
Parameters
a : array_like
Array containing elements to clip.

a_min : scalar or array_like
Minimum value.

a_max : scalar or array_like
Maximum value. If a_min or a_max are array_like, then they will be broadcasted to the shape of a.

out : ndarray, optional
The results will be placed in this array. It may be the input array for in-place clipping. out must be of the right shape to hold the output. Its type is preserved.

Returns
clipped_array : ndarray
An array with the elements of a, but where values < a_min are replaced with a_min, and those > a_max with a_max.

See Also:
numpy.doc.ufuncs
Section “Output arguments”

Examples
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> np.clip(a, [3,4,1,1,4,4,4,4,4,4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])

sqrt ()
Return the positive square-root of an array, element-wise.

Parameters
x : array_like
The square root of each element in this array is calculated.

Returns
y : ndarray
An array of the same shape as x, containing the square-root of each element in x. If any element in x is complex, a complex array is returned. If all of the elements of x are real, negative elements return numpy.nan elements.

See Also:
numpy.lib.scimath.sqrt
A version which returns complex numbers when given negative reals.

Notes

sqrt has a branch cut \([-\infty, 0)\) and is continuous from above on it.

Examples

```python
>>> np.sqrt([1, 4, 9])
array([ 1., 2., 3.])
```

```python
>>> np.sqrt([4, -1, -3+4j])
array([ 2.+0.j, 0.+1.j, 1.+2.j])
```

```python
>>> np.sqrt([4, -1, numpy.inf])
array([ 2., NaN, Inf])
```

square()
Return the element-wise square of the input.

Parameters

- x : array_like
  Input data.

Returns

- out : ndarray
  Element-wise \(x \times x\), of the same shape and dtype as \(x\). Returns scalar if \(x\) is a scalar.

See Also:

numpy.linalg.matrix_power, sqrt, power

Examples

```python
>>> np.square([-1j, 1])
array([-1.-0.j, 1.+0.j])
```

absolute()
Calculate the absolute value element-wise.

Parameters

- x : array_like
  Input array.

Returns

- res : ndarray
  An ndarray containing the absolute value of each element in \(x\). For complex input, \(a + ib\), the absolute value is \(\sqrt{a^2 + b^2}\).

Examples

```python
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2,  1.2])
```
```python
>>> np.absolute(1.2 + 1j)
1.5620499351813308
```

Plot the function over \([-10, 10]\):
```python
>>> import matplotlib.pyplot as plt

>>> x = np.linspace(-10, 10, 101)
>>> plt.plot(x, np.absolute(x))
>>> plt.show()
```

Plot the function over the complex plane:
```python
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10])
>>> plt.show()
```

**fabs** ()
Compute the absolute values elementwise.
This function returns the absolute values (positive magnitude) of the data in \(x\). Complex values are not handled, use `absolute` to find the absolute values of complex data.

**Parameters**
- **x**: array_like
  The array of numbers for which the absolute values are required. If \(x\) is a scalar, the result \(y\) will also be a scalar.

- **out**: ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

**Returns**
- **y**: [ndarray, scalar]
  The absolute values of \(x\), the returned values are always floats.
See Also:

**absolute**

Absolute values including *complex* types.

**Examples**

```python
>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([ 1.2, 1.2])
```

**sign**

Returns an element-wise indication of the sign of a number.

The *sign* function returns \(-1\) if \(x < 0\), \(0\) if \(x == 0\), \(1\) if \(x > 0\).

**Parameters**

- `x`: array_like
  
  Input values.

**Returns**

- `y`: ndarray
  
  The sign of \(x\).

**Examples**

```python
>>> np.sign([-5., 4.5])
array([-1., 1.])
>>> np.sign(0)
0
```

**maximum**

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a nan, then that element is returned. If both elements are nans then the first is returned. The latter distinction is important for complex nans, which are defined as at least one of the real or imaginary parts being a nan. The net effect is that nans are propagated.

**Parameters**

- `x1, x2`: array_like
  
  The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

**Returns**

- `y`: [ndarray, scalar]
  
  The maximum of \(x1\) and \(x2\), element-wise. Returns scalar if both \(x1\) and \(x2\) are scalars.

**See Also:**

**minimum**

- `element-wise minimum`

**fmax**

- `element-wise maximum that ignores nans unless both inputs are nans`
\textbf{fmin}

Element-wise minimum that ignores nans unless both inputs are nans.

\textbf{Notes}

Equivalent to \texttt{np.where(x1 > x2, x1, x2)} but faster and does proper broadcasting.

\textbf{Examples}

\begin{verbatim}
>>> np.maximum([2, 3, 4], [1, 5, 2])
array([2, 5, 4])
>>> np.maximum(np.eye(2), [0.5, 2])
array([[1., 2.],
       [0.5, 2.]])
>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([ NaN, NaN, NaN])
>>> np.maximum(np.Inf, 1)
inf
\end{verbatim}

\textbf{minimum()}

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a nan, then that element is returned. If both elements are nans then the first is returned. The latter distinction is important for complex nans, which are defined as at least one of the real or imaginary parts being a nan. The net effect is that nans are propagated.

\textbf{Parameters}

\begin{verbatim}
x1, x2 : array_like
\end{verbatim}

The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

\textbf{Returns}

\begin{verbatim}
y : {ndarray, scalar}
\end{verbatim}

The minimum of \texttt{x1} and \texttt{x2}, element-wise. Returns scalar if both \texttt{x1} and \texttt{x2} are scalars.

\textbf{See Also:}

\textbf{maximum}

Element-wise minimum that propagates nans.

\textbf{fmax}

Element-wise maximum that ignores nans unless both inputs are nans.

\textbf{fmin}

Element-wise minimum that ignores nans unless both inputs are nans.

\textbf{Notes}

The minimum is equivalent to \texttt{np.where(x1 <= x2, x1, x2)} when neither \texttt{x1} nor \texttt{x2} are nans, but it is faster and does proper broadcasting.
Examples

```python
>>> np.minimum([2, 3, 4], [1, 5, 2])
array([1, 3, 2])
```

```python
>>> np.minimum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 0.5,  0. ],
        [ 0. ,  1. ]])
```

```python
>>> np.minimum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([NaN, NaN, NaN])
```

**nan_to_num** *(x)*

Replace nan with zero and inf with finite numbers.

Returns an array or scalar replacing Not a Number (NaN) with zero, (positive) infinity with a very large number and negative infinity with a very small (or negative) number.

**Parameters**

- **x**: array_like
  Input data.

**Returns**

- **out**: ndarray, float
  Array with the same shape as x and dtype of the element in x with the greatest precision. NaN is replaced by zero, and infinity (-infinity) is replaced by the largest (smallest or most negative) floating point value that fits in the output dtype. All finite numbers are upcast to the output dtype (default float64).

**See Also:**

- **isinf**
  Shows which elements are negative or negative infinity.
- **isneginf**
  Shows which elements are negative infinity.
- **isposinf**
  Shows which elements are positive infinity.
- **isnan**
  Shows which elements are Not a Number (NaN).
- **isfinite**
  Shows which elements are finite (not NaN, not infinity)

**Notes**

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

**Examples**

```python
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
>>> np.nan_to_num(x)
array([-1.28000000e+02,  1.28000000e+02, -1.28000000e+02,  0.00000000e+00,
       -1.79769313e+308, -1.79769313e+308,  0.00000000e+00])
```

real_if_close

If complex input returns a real array if complex parts are close to zero.

"Close to zero" is defined as $tol \times \text{(machine epsilon of the type for } a\text{)}$.

**Parameters**

- $a$ : array_like
  - Input array.
- $tol$ : float
  - Tolerance in machine epsilons for the complex part of the elements in the array.

**Returns**

- $out$ : ndarray
  - If $a$ is real, the type of $a$ is used for the output. If $a$ has complex elements, the returned type is float.

**See Also:**

real, imag, angle

**Notes**

Machine epsilon varies from machine to machine and between data types but Python floats on most platforms have a machine epsilon equal to 2.2204460492503131e-16. You can use ‘np.finfo(np.float).eps’ to print out the machine epsilon for floats.

**Examples**

```python
>>> np.finfo(np.float).eps
2.2204460492503131e-16

>>> np.real_if_close([2.1 + 4e-14j], tol=1000)
array([ 2.1])
>>> np.real_if_close([2.1 + 4e-13j], tol=1000)
array([ 2.1 +4.00000000e-13j])
```

interp

One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function with given values at discrete data-points.

**Parameters**

- $x$ : array_like
  - The x-coordinates of the interpolated values.
- $xp$ : 1-D sequence of floats
  - The x-coordinates of the data points, must be increasing.
- $fp$ : 1-D sequence of floats
  - The y-coordinates of the data points, same length as $xp$.
- left : float, optional
  - Value to return for $x$, default is $fp[0]$.
- right : float, optional
  - Value to return for $x > xp[-1]$, defaults is $fp[-1]$.
Returns

\[ y : \text{float, ndarray} \]

The interpolated values, same shape as \( x \).

Raises

ValueError :

If \( xp \) and \( fp \) have different length

Notes

Does not check that the x-coordinate sequence \( xp \) is increasing. If \( xp \) is not increasing, the results are nonsense. A simple check for increasingness is:

\[
\text{np.all(np.diff(xp) > 0)}
\]

Examples

```python
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([ 3. , 3. , 2.5, 0.56, 0. ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0
```

Plot an interpolant to the sine function:

```python
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
>>> plt.plot(xvals, yinterp, '-x')
>>> plt.show()
```
3.14 Functional programming

| apply_along_axis (func1d, axis, arr, ...) | Apply a function to 1-D slices along the given axis. |
| apply_over_axes (func, a, axes) | Apply a function repeatedly over multiple axes. |
| vectorize (pyfunc[, otypes, doc]) | Generalized function class. |
| frompyfunc (func, nin, nout) | Takes an arbitrary Python function and returns a Numpy ufunc. |
| piecewise (x, condlist, funclist, ...) | Evaluate a piecewise-defined function. |

**apply_along_axis** *(func1d, axis, arr, *args)*

Apply a function to 1-D slices along the given axis.

Execute *func1d(a, *args)* where *func1d* operates on 1-D arrays and *a* is a 1-D slice of *arr* along *axis*.

**Parameters**

- **func1d**: function
  
  This function should accept 1-D arrays. It is applied to 1-D slices of *arr* along the specified axis.

- **axis**: integer
  
  Axis along which *arr* is sliced.

- **arr**: ndarray
  
  Input array.

- **args**: any
  
  Additional arguments to *func1d*.

**Returns**

- **outarr**: ndarray
  
  The output array. The shape of *outarr* is identical to the shape of *arr*, except along the *axis* dimension, where the length of *outarr* is equal to the size of the return value of *func1d*. If *func1d* returns a scalar *outarr* will have one fewer dimensions than *arr*.

**See Also**:
apply_over_axes

Apply a function repeatedly over multiple axes.

Examples

```python
>>> def my_func(a):
...     return (a[0] + a[-1]) * 0.5
... b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2., 5., 8.])
```

For a function that doesn’t return a scalar, the number of dimensions in `outarr` is the same as `arr`.

```python
>>> def new_func(a):
...     return a * 0.5
... b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(new_func, 0, b)
array([[ 0.5, 1. , 1.5],
       [ 2. , 2.5, 3. ],
       [ 3.5, 4. , 4.5]])
```

apply_over_axes (`func`, `a`, `axes`)  

Apply a function repeatedly over multiple axes.

`func` is called as `res = func(a, axis)`, where `axis` is the first element of `axes`. The result `res` of the function call must have either the same dimensions as `a` or one less dimension. If `res` has one less dimension than `a`, a dimension is inserted before `axis`. The call to `func` is then repeated for each axis in `axes`, with `res` as the first argument.

**Parameters**

- `func` : function
  
  This function must take two arguments, `func(a, axis)`.
  
  - `a` : ndarray
    
    Input array.
  
  - `axes` : array_like
    
    Axes over which `func` is applied, the elements must be integers.

**Returns**

- `val` : ndarray
  
  The output array. The number of dimensions is the same as `a`, but the shape can be different. This depends on whether `func` changes the shape of its output with respect to its input.

**See Also:**

apply_along_axis

Apply a function to 1-D slices of an array along the given axis.
Examples

```python
>>> a = np.arange(24).reshape(2,3,4)
>>> a
array(
    [[[ 0,  1,  2,  3],
      [ 4,  5,  6,  7],
      [ 8,  9, 10, 11]],
     [[12, 13, 14, 15],
      [16, 17, 18, 19],
      [20, 21, 22, 23]]))
```

Sum over axes 0 and 2. The result has same number of dimensions as the original array:

```python
>>> np.apply_over_axes(np.sum, a, [0,2])
array(
    [[[ 60],
      [ 92],
      [124]]])
```

class `vectorize`(pyfunc, otypes=", `doc=None)

Generalized function class.

Define a vectorized function which takes a nested sequence of objects or numpy arrays as inputs and returns a numpy array as output. The vectorized function evaluates `pyfunc` over successive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

The data type of the output of `vectorize` is determined by calling the function with the first element of the input. This can be avoided by specifying the `otypes` argument.

Parameters

- `pyfunc`: callable
  A python function or method.
- `otypes`: str or list of dtypes, optional
  The output data type. It must be specified as either a string of typecode characters or a list of data type specifiers. There should be one data type specifier for each output.
- `doc`: str, optional
  The docstring for the function. If None, the docstring will be the `pyfunc` one.

Examples

```python
>>> def myfunc(a, b):
...     """Return a-b if a>b, otherwise return a+b""
...     if a > b:
...         return a - b
...     else:
...         return a + b

>>> vfunc = np.vectorize(myfunc)
>>> vfunc([1, 2, 3, 4], 2)
array([3, 4, 1, 2])
```

The docstring is taken from the input function to `vectorize` unless it is specified

```python
>>> vfunc.__doc__
'Return a-b if a>b, otherwise return a+b'
```
>> vfunc = np.vectorize(myfunc, doc='Vectorized myfunc')
>> vfunc.__doc__
'Vectorized myfunc'

The output type is determined by evaluating the first element of the input, unless it is specified

>> out = vfunc([1, 2, 3, 4], 2)
>> type(out[0])
<type 'numpy.int32'>
>> vfunc = np.vectorize(myfunc, otypes=[np.float])
>> out = vfunc([1, 2, 3, 4], 2)
>> type(out[0])
<type 'numpy.float64'>

frompyfunc()
Takes an arbitrary Python function and returns a Numpy ufunc.
Can be used, for example, to add broadcasting to a built-in Python function (see Examples section).

Parameters
- **func**: Python function object
  - An arbitrary Python function.
- **nin**: int
  - The number of input arguments.
- **nout**: int
  - The number of objects returned by func.

Returns
- **out**: ufunc
  - Returns a Numpy universal function (ufunc) object.

Notes
The returned ufunc always returns PyObject arrays.

Examples
Use frompyfunc to add broadcasting to the Python function oct:

>>> oct_array = np.frompyfunc(oct, 1, 1)
>>> oct_array(np.array((10, 30, 100)))
array([012, 036, 0144], dtype=object)
>>> np.array((oct(10), oct(30), oct(100)))  # for comparison
array(['012', '036', '0144'],
      dtype='|S4')

piecewise(x, condlist, funclist, *args, **kw)
Evaluate a piecewise-defined function.
Given a set of conditions and corresponding functions, evaluate each function on the input data wherever its condition is true.

Parameters
- **x**: ndarray
  - The input domain.
**condlist**: list of bool arrays

Each boolean array corresponds to a function in *funclist*. Wherever *condlist[i]* is True, *funclist[i](x)* is used as the output value.

Each boolean array in *condlist* selects a piece of *x*, and should therefore be of the same shape as *x*.

The length of *condlist* must correspond to that of *funclist*. If one extra function is given, i.e. if `len(funclist) - len(condlist) == 1`, then that extra function is the default value, used wherever all conditions are false.

**funclist**: list of callables, f(x,*args,**kw), or scalars

Each function is evaluated over *x* wherever its corresponding condition is True. It should take an array as input and give an array or a scalar value as output. If, instead of a callable, a scalar is provided then a constant function (*lambda x: scalar*) is assumed.

**args**: tuple, optional

Any further arguments given to *piecewise* are passed to the functions upon execution, i.e., if called *piecewise(..., ..., 1, 'a')*, then each function is called as *f(x, 1, 'a').*

**kw**: dict, optional

Keyword arguments used in calling *piecewise* are passed to the functions upon execution, i.e., if called *piecewise(..., ..., lambda=1)*, then each function is called as *f(x, lambda=1)*.

**Returns**

**out**: ndarray

The output is the same shape and type as *x* and is found by calling the functions in *funclist* on the appropriate portions of *x*, as defined by the boolean arrays in *condlist*. Portions not covered by any condition have undefined values.

**See Also:**

*choose*, *select*, *where*

**Notes**

This is similar to *choose* or *select*, except that functions are evaluated on elements of *x* that satisfy the corresponding condition from *condlist*.

The result is:

```
|--
|funclist[0](x[condlist[0]])
out = |funclist[1](x[condlist[1]])
|...
|funclist[n2](x[condlist[n2]])
|--
```

**Examples**

Define the sigma function, which is -1 for *x* < 0 and +1 for *x* >= 0.
>>> x = np.arange(6) - 2.5
>>> np.piecewise(x, [x < 0, x >= 0], [-1, 1])
array([[-1., -1., -1., 1., 1., 1.]])

Define the absolute value, which is \(-x\) for \(x < 0\) and \(x\) for \(x \geq 0\).

```python
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([ 2.5, 1.5, 0.5, 0.5, 1.5, 2.5])
```

## 3.15 Polynomials

### 3.15.1 Basics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poly1d (c_or_r, r=0, variable=None)</td>
<td>A one-dimensional polynomial class.</td>
</tr>
<tr>
<td>polyval (p, x)</td>
<td>Evaluate a polynomial at specific values.</td>
</tr>
<tr>
<td>poly (seq_of_zeros)</td>
<td>Find the coefficients of a polynomial with the given sequence of roots.</td>
</tr>
<tr>
<td>roots (p)</td>
<td>Return the roots of a polynomial with coefficients given in p.</td>
</tr>
</tbody>
</table>

**class poly1d (c_or_r, r=0, variable=None)**

A one-dimensional polynomial class.

A convenience class, used to encapsulate “natural” operations on polynomials so that said operations may take on their customary form in code (see Examples).

**Parameters**

- **c_or_r**: array_like
  
  The polynomial’s coefficients, in decreasing powers, or if the value of the second parameter is True, the polynomial’s roots (values where the polynomial evaluates to 0). For example, `poly1d([1, 2, 3])` returns an object that represents \(x^2 + 2x + 3\), whereas `poly1d([1, 2, 3], True)` returns one that represents \((x - 1)(x - 2)(x - 3) = x^3 - 6x^2 + 11x - 6\).

- **r**: bool, optional
  
  If True, `c_or_r` specifies the polynomial’s roots; the default is False.

- **variable**: str, optional
  
  Changes the variable used when printing `p` from `x` to `variable` (see Examples).

**Examples**

Construct the polynomial \(x^2 + 2x + 3\):

```python
>>> p = np.poly1d([1, 2, 3])
>>> print np.poly1d(p)
2
1 x + 2 x + 3
```

Evaluate the polynomial at \(x = 0.5\):

```python
>>> p(0.5)
4.25
```

Find the roots:
>>> p.r
array([-1.+1.41421356j, -1.-1.41421356j])
>>> p(p.r)
array([-4.44089210e-16+0.j, -4.44089210e-16+0.j]) # i.e., (0, 0)

Show the coefficients:

>>> p.c
array([1, 2, 3])

Display the order (the leading zero-coefficients are removed):

>>> p.order
2

Show the coefficient of the k-th power in the polynomial (which is equivalent to \( p.c[-(i+1)] \)):

>>> p[1]
2

Polynomials can be added, subtracted, multiplied, and divided (returns quotient and remainder):

>>> p * p
poly1d([1, 4, 10, 12, 9])

>>> (p**3 + 4) / p
(poly1d([1, 4, 10, 12, 9]), poly1d([4]))

`asarray(p)` gives the coefficient array, so polynomials can be used in all functions that accept arrays:

>>> p**2 # square of polynomial
poly1d([1, 4, 10, 12, 9])

>>> np.square(p) # square of individual coefficients
array([1, 4, 9])

The variable used in the string representation of \( p \) can be modified, using the `variable` parameter:

>>> p = np.poly1d([1,2,3], variable='z')
>>> print p
  2
  1 z + 2 z + 3

Construct a polynomial from its roots:

>>> np.poly1d([1, 2], True)
poly1d([1, -3, 2])

This is the same polynomial as obtained by:

>>> np.poly1d([1, -1]) * np.poly1d([1, -2])
poly1d([1, -3, 2])"
Attributes

Methods

polyval \((p, x)\)
Evaluate a polynomial at specific values.

If \(p\) is of length \(N\), this function returns the value:

\[ p[0] \times x^{(N-1)} + p[1] \times x^{(N-2)} + \ldots + p[N-2] \times x + p[N-1] \]

If \(x\) is a sequence, then \(p(x)\) is returned for each element of \(x\). If \(x\) is another polynomial then the composite polynomial \(p(x(t))\) is returned.

Parameters

- \(p\) : array_like or poly1d object
  1D array of polynomial coefficients (including coefficients equal to zero) from highest degree to the constant term, or an instance of poly1d.

- \(x\) : array_like or poly1d object
  A number, a 1D array of numbers, or an instance of poly1d, “at” which to evaluate \(p\).

Returns

- \(values\) : ndarray or poly1d
  If \(x\) is a poly1d instance, the result is the composition of the two polynomials, i.e., \(x\) is “substituted” in \(p\) and the simplified result is returned. In addition, the type of \(x\) - array_like or poly1d - governs the type of the output: \(x\) array_like \(\Rightarrow\) \(values\) array_like, \(x\) a poly1d object \(\Rightarrow\) \(values\) is also.

See Also:

poly1d
A polynomial class.

Notes

Horner’s scheme \([R68]\) is used to evaluate the polynomial. Even so, for polynomials of high degree the values may be inaccurate due to rounding errors. Use carefully.

References

[R68]

Examples

```python
>>> np.polyval([3,0,1], 5)  # 3 \times 5^2 + 0 \times 5 + 1
76
>>> np.polyval([3,0,1], np.poly1d(5))
poly1d([ 76.])
>>> np.polyval(np.poly1d([3,0,1]), 5)
76
>>> np.polyval(np.poly1d([3,0,1]), np.poly1d(5))
poly1d([ 76.])
```
poly(seq_of_zeros)

Find the coefficients of a polynomial with the given sequence of roots.

Returns the coefficients of the polynomial whose leading coefficient is one for the given sequence of zeros (multiple roots must be included in the sequence as many times as their multiplicity; see Examples). A square matrix (or array, which will be treated as a matrix) can also be given, in which case the coefficients of the characteristic polynomial of the matrix are returned.

Parameters
seq_of_zeros : array_like, shape (N,) or (N, N)
A sequence of polynomial roots, or a square array or matrix object.

Returns
c : ndarray
1D array of polynomial coefficients from highest to lowest degree:
\[ c[0] \times x^{N} + c[1] \times x^{N-1} + \ldots + c[N-1] \times x + c[N] \]
where \( c[0] \) always equals 1.

Raises
ValueError :
If input is the wrong shape (the input must be a 1-D or square 2-D array).

See Also:
polyval
Evaluate a polynomial at a point.

roots
Return the roots of a polynomial.

polyfit
Least squares polynomial fit.

poly1d
A one-dimensional polynomial class.

Notes
Specifying the roots of a polynomial still leaves one degree of freedom, typically represented by an undetermined leading coefficient. [R64] In the case of this function, that coefficient - the first one in the returned array - is always taken as one. (If for some reason you have one other point, the only automatic way presently to leverage that information is to use polyfit.)

The characteristic polynomial, \( p_a(t) \), of an \( n \)-by-\( n \) matrix \( A \) is given by
\[ p_a(t) = \det(tI - A), \]
where \( I \) is the \( n \)-by-\( n \) identity matrix. [R65]

References
[R64], [R65]

Examples
Given a sequence of a polynomial’s zeros:
>>> np.poly((0, 0, 0)) # Multiple root example
array([1, 0, 0, 0]) # i.e., z**3 + 0*z**2 + 0*z + 0

>>> np.poly((-1./2, 0, 1./2))
array([1.0, 0.0, -0.25, 0.0]) # z**3 - z/4

>>> np.poly((np.random.random(1)[0], 0, np.random.random(1)[0]))
array([1.0, -0.77086955, 0.08618131, 0.0])

Given a square array object:

>>> P = np.array([[0, 1./3], [-1./2, 0]])

>>> np.poly(P)
array([1.0, 0.0, 0.16666667])

Or a square matrix object:

>>> np.poly(np.matrix(P))
array([1.0, 0.0, 0.16666667])

Note how in all cases the leading coefficient is always 1.

roots(p)
Return the roots of a polynomial with coefficients given in p.

The values in the rank-1 array p are coefficients of a polynomial. If the length of p is n+1 then the polynomial is described by p[0] * x**n + p[1] * x**(n-1) + ... + p[n-1]*x + p[n]

Parameters
   p : array_like of shape(M,)
      Rank-1 array of polynomial co-efficients.

Returns
   out : ndarray
      An array containing the complex roots of the polynomial.

Raises
   ValueError: :
      When p cannot be converted to a rank-1 array.

See Also:

poly
Find the coefficients of a polynomial with a given sequence of roots.

polyval
Evaluate a polynomial at a point.

polyfit
Least squares polynomial fit.

poly1d
A one-dimensional polynomial class.

Notes
The algorithm relies on computing the eigenvalues of the companion matrix [R135].
References

[R135]

Examples

```python
>>> coeff = [3.2, 2, 1]
>>> np.roots(coeff)
array([-0.3125+0.46351241j, -0.3125-0.46351241j])
```

3.15.2 Fitting

```
polyfit (x, y, deg[, rcond, full])  Least squares polynomial fit.
```

```
polyfit (x, y, deg=deg, rcond=none, full=False)
```

Least squares polynomial fit.

Fit a polynomial \( p(x) = p[0] \times x^{\text{deg}} + \ldots + p[\text{deg}] \) of degree `deg` to points \((x, y)\). Returns a vector of coefficients \( p \) that minimizes the squared error.

**Parameters**

- **x**: array_like, shape (M,)
  - x-coordinates of the M sample points \((x[i], y[i])\).
- **y**: array_like, shape (M,) or (M, K)
  - y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.
- **deg**: int
  - Degree of the fitting polynomial
- **rcond**: float, optional
  - Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \( \text{len}(x) \times \text{eps} \), where eps is the relative precision of the float type, about 2e-16 in most cases.
- **full**: bool, optional
  - Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**Returns**

- **p**: ndarray, shape (M,) or (M, K)
  - Polynomial coefficients, highest power first. If y was 2-D, the coefficients for k-th data set are in \( p[:, k] \).
- **residuals**, **rank**, **singular_values**, **rcond**: present only if full = True
  - Residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of rcond. For more details, see `linalg.lstsq`.

See Also:
polyval
Computes polynomial values.

linalg.lstsq
Computes a least-squares fit.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes
The solution minimizes the squared error
\[ E = \sum_{j=0}^{k} |p(x_j) - y_j|^2 \]
in the equations:
\[ x[0]^{\ast n} \ast p[n] + \ldots + x[0] \ast p[1] + p[0] = y[0] \]
\[ \ldots \]
\[ x[k]^{\ast n} \ast p[n] + \ldots + x[k] \ast p[1] + p[0] = y[k] \]
The coefficient matrix of the coefficients \( p \) is a Vandermonde matrix.

polyfit issues a RankWarning when the least-squares fit is badly conditioned. This implies that the best fit is not well-defined due to numerical error. The results may be improved by lowering the polynomial degree or by replacing \( x \) by \( x - x.\text{mean}() \). The rcond parameter can also be set to a value smaller than its default, but the resulting fit may be spurious: including contributions from the small singular values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned when the degree of the polynomial is large or the interval of sample points is badly centered. The quality of the fit should always be checked in these cases. When polynomial fits are not satisfactory, splines may be a good alternative.

References
[R66], [R67]

Examples

```python
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206, 1.69312169, -0.03968254])
```

It is convenient to use poly1d objects for dealing with polynomials:

```python
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:
```python
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```python
>>> import matplotlib.pyplot as plt

>>> xp = np.linspace(-2, 6, 100)

>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')

>>> plt.ylim(-2,2)

>>> plt.show()
```

3.15.3 Calculus

<table>
<thead>
<tr>
<th>polyder (p, m)</th>
<th>Return the derivative of the specified order of a polynomial.</th>
</tr>
</thead>
<tbody>
<tr>
<td>polyint (p, m, k)</td>
<td>Return an antiderivative (indefinite integral) of a polynomial.</td>
</tr>
</tbody>
</table>

polyder (p, m=1)

Return the derivative of the specified order of a polynomial.

Parameters

- **p**: poly1d or sequence
  - Polynomial to differentiate. A sequence is interpreted as polynomial coefficients, see `poly1d`.
- **m**: int, optional
  - Order of differentiation (default: 1)

Returns

- **der**: poly1d
A new polynomial representing the derivative.

See Also:

polyint
   Anti-derivative of a polynomial.

dolyld
   Class for one-dimensional polynomials.

Examples

The derivative of the polynomial \( x^3 + x^2 + x + 1 \) is:

```python
>>> p = np.poly1d([1,1,1,1])
>>> p2 = np.polyder(p)
>>> p2
poly1d([3, 2, 1])
```

which evaluates to:

```python
>>> p2(2.)
17.0
```

We can verify this, approximating the derivative with \( (f(x + h) - f(x))/h \):

```python
>>> (p(2. + 0.001) - p(2.)) / 0.001
17.007000999997857
```

The fourth-order derivative of a 3rd-order polynomial is zero:

```python
>>> np.polyder(p, 2)
poly1d([6, 2])
>>> np.polyder(p, 3)
poly1d([6])
>>> np.polyder(p, 4)
poly1d([0.])
```

polyint
   \( p, m=1, k=None \)

Return an antiderivative (indefinite integral) of a polynomial.

The returned order \( m \) antiderivative \( P \) of polynomial \( p \) satisfies \( \frac{d^m}{dx^m} P(x) = p(x) \) and is defined up to \( m - 1 \) integration constants \( k \). The constants determine the low-order polynomial part

\[
    \frac{k_{m-1}}{0!} x^0 + \ldots + \frac{k_0}{(m-1)!} x^{m-1}
\]

of \( P \) so that \( P^{(j)}(0) = k_{m-j-1} \).

Parameters

- \( p \) : {array_like, poly1d}
  Polynomial to differentiate. A sequence is interpreted as polynomial coefficients, see poly1d.

- \( m \) : int, optional
  Order of the antiderivative. (Default: 1)
**k**: {None, list of \( m \) scalars, scalar}, optional

Integration constants. They are given in the order of integration: those corresponding to highest-order terms come first.

If None (default), all constants are assumed to be zero. If \( m = 1 \), a single scalar can be given instead of a list.

**See Also:**

*polyder*  
derivative of a polynomial

*poly1d.integ*  
equivalent method

**Examples**

The defining property of the antiderivative:

```python
gp = np.poly1d([1, 1, 1])
P = np.polyint(p)
poly1d([0.33333333, 0.5, 1., 0.])
np.polyder(P) == p
True
```

The integration constants default to zero, but can be specified:

```python
P = np.polyint(p, 3)
P(0)
0.0
np.polyder(P)(0)
0.0
np.polyder(P, 2)(0)
0.0
P = np.polyint(p, 3, k=[6, 5, 3])
P
poly1d([0.01666667, 0.04166667, 0.16666667, 3., 5., 3.])
```

Note that \( 3 = 6 / 2! \), and that the constants are given in the order of integrations. Constant of the highest-order polynomial term comes first:

```python
np.polyder(P, 2)(0)
6.0
np.polyder(P, 1)(0)
5.0
P(0)
3.0
```

### 3.15.4 Arithmetic

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>polyadd(a1, a2)</td>
<td>Find the sum of two polynomials.</td>
</tr>
<tr>
<td>polydiv(u, v)</td>
<td>Returns the quotient and remainder of polynomial division.</td>
</tr>
<tr>
<td>polymul(a1, a2)</td>
<td>Find the product of two polynomials.</td>
</tr>
<tr>
<td>polysub(a1, a2)</td>
<td>Difference (subtraction) of two polynomials.</td>
</tr>
</tbody>
</table>
polyadd \((a1, a2)\)
Find the sum of two polynomials.

Returns the polynomial resulting from the sum of two input polynomials. Each input must be either a poly1d object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

Parameters
\(a1, a2\) : array_like or poly1d object
Input polynomials.

Returns
out : ndarray or poly1d object
The sum of the inputs. If either input is a poly1d object, then the output is also a poly1d object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

See Also:

poly1d
A one-dimensional polynomial class.

poly, polyadd, polyder, polydiv, polyfit, polyint, polysub, polyval

Examples

```python
>>> np.polyadd([1, 2], [9, 5, 4])
array([9, 6, 6])
```

Using poly1d objects:

```python
>>> p1 = np.poly1d([1, 2])
>>> p2 = np.poly1d([9, 5, 4])
>>> print p1
1 x + 2
>>> print p2
2
9 x + 5 x + 4
>>> print np.polyadd(p1, p2)
2
9 x + 6 x + 6
```

polydiv \((u, v)\)
Returns the quotient and remainder of polynomial division.

The input arrays are the coefficients (including any coefficients equal to zero) of the “numerator” (dividend) and “denominator” (divisor) polynomials, respectively.

Parameters
\(u\) : array_like or poly1d
Dividend polynomial's coefficients.

\(v\) : array_like or poly1d
Divisor polynomial's coefficients.

Returns
q : ndarray
Coefficients, including those equal to zero, of the quotient.
polymul

Find the product of two polynomials.

Finds the polynomial resulting from the multiplication of the two input polynomials. Each input must be either a poly1d object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

**Parameters**

- `a1, a2`: array_like or poly1d object
  
  Input polynomials.

**Returns**

- `out`: ndarray or poly1d object
  
  The polynomial resulting from the multiplication of the inputs. If either inputs is a poly1d object, then the output is also a poly1d object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

**Examples**

```python
>>> np.polymul([1, 2, 3], [9, 5, 1])
array([ 9, 23, 38, 17,  3])
```

Using poly1d objects:
>>> p1 = np.poly1d([1, 2, 3])
>>> p2 = np.poly1d([9, 5, 1])
>>> print p1
  2
1 x + 2 x + 3
>>> print p2
  2
9 x + 5 x + 1
>>> print np.polymul(p1, p2)
  4 3 2
9 x + 23 x + 38 x + 17 x + 3

polysub (a1, a2)
Difference (subtraction) of two polynomials.

Given two polynomials a1 and a2, returns a1 - a2. a1 and a2 can be either
array_like sequences of the polynomials’ coefficients (including coefficients
equal to zero), or poly1d objects.

Parameters
  a1, a2 : array_like or poly1d
    Minuend and subtrahend polynomials, respectively.

Returns
  out : ndarray or poly1d
    Array or poly1d object of the difference polynomial’s coefficients.

See Also:
polyval, polydiv, polymul, polyadd

Examples

\[(2x^2 + 10x - 2) - (3x^2 + 10x - 4) = (-x^2 + 2)\]

>>> np.polysub([2, 10, -2], [3, 10, -4])
array([-1,  0,  2])

3.15.5 Warnings

RankWarning | Issued by polyfit when the Vandermonde matrix is rank deficient.

exception RankWarning
  Issued by polyfit when the Vandermonde matrix is rank deficient.

   For more information, a way to suppress the warning, and an example of RankWarning being issued, see polyfit.
3.16 Financial functions

3.16.1 Simple financial functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fv</code></td>
<td>Compute the future value.</td>
</tr>
<tr>
<td><code>pv</code></td>
<td>Compute the present value.</td>
</tr>
<tr>
<td><code>npv</code></td>
<td>Returns the NPV (Net Present Value) of a cash flow series.</td>
</tr>
<tr>
<td><code>pmt</code></td>
<td>Compute the payment against loan principal plus interest.</td>
</tr>
<tr>
<td><code>ipmt</code></td>
<td>Not implemented. Compute the payment portion for loan principal.</td>
</tr>
<tr>
<td><code>irr</code></td>
<td>Return the Internal Rate of Return (IRR).</td>
</tr>
<tr>
<td><code>mirr</code></td>
<td>Modified internal rate of return.</td>
</tr>
<tr>
<td><code>nper</code></td>
<td>Compute the number of periods.</td>
</tr>
<tr>
<td><code>rate</code></td>
<td>Compute the rate of interest per period.</td>
</tr>
</tbody>
</table>

### `fv` (rate, nper, pmt, pv[, when])

Compute the future value.

**Given:**

- a present value, `pv`
- an interest rate compounded once per period, of which there are
  - `nper` total
- a (fixed) payment, `pmt`, paid either
  - at the beginning (`when` = ‘begin’, 1)) or the end (`when` = ‘end’, 0)) of each period

**Return:**

the value at the end of the `nper` periods

**Parameters**

- `rate`: scalar or array_like of shape(M, )
  - Rate of interest as decimal (not per cent) per period
- `nper`: scalar or array_like of shape(M, )
  - Number of compounding periods
- `pmt`: scalar or array_like of shape(M, )
  - Payment
- `pv`: scalar or array_like of shape(M, )
  - Present value
- `when`: {‘begin’, 1}, {‘end’, 0}, string, int, optional
  - When payments are due (`begin` (1) or `end` (0)). Defaults to {‘end’, 0}.

**Returns**

- `out`: ndarray
  - Future values. If all input is scalar, returns a scalar float. If any input is array_like, returns future values for each input element. If multiple inputs are array_like, they all must have the same shape.
Notes

The future value is computed by solving the equation:

\[ fv + pv \times (1+rate)^{nper} + pmt \times \frac{(1 + rate \times when) / rate - (1 + rate)^{nper} - 1}{rate} == 0 \]

or, when rate == 0:

\[ fv + pv + pmt \times nper == 0 \]

References

[WRW]

Examples

What is the future value after 10 years of saving $100 now, with an additional monthly savings of $100. Assume the interest rate is 5% (annually) compounded monthly?

```python
>>> np.fv(0.05/12, 10*12, -100, -100)
15692.928894335748
```

By convention, the negative sign represents cash flow out (i.e. money not available today). Thus, saving $100 a month at 5% annual interest leads to $15,692.93 available to spend in 10 years.

If any input is array_like, returns an array of equal shape. Let’s compare different interest rates from the example above.

```python
>>> a = np.array((0.05, 0.06, 0.07))/12
>>> np.fv(a, 10*12, -100, -100)
array([ 15692.92889434, 16569.87435405, 17509.44688102])
```

\[ pv (rate, nper, pmt, fv=0.0, when='end') \]

Compute the present value.

Given:

- a future value, fv
- an interest rate compounded once per period, of which there are
- nper total
- a (fixed) payment, pmt, paid either
- at the beginning (when = {'begin', 1}) or the end (when = {'end', 0}) of each period

Return:

the value now

Parameters

rate : array_like
    Rate of interest (per period)

nper : array_like
    Number of compounding periods
pmt : array_like
    Payment
fv : array_like, optional
    Future value
when : {{‘begin’, 1}, {‘end’, 0}}, {string, int}, optional
    When payments are due (‘begin’ (1) or ‘end’ (0))

Returns
out : ndarray, float
    Present value of a series of payments or investments.

Notes
The present value is computed by solving the equation:

\[ \text{fv} + \text{pv} \times (1 + \text{rate})^{\text{nper}} + \]
\[ \text{pmt} \times (1 + \text{rate} \times \text{when}) / \text{rate} \times ((1 + \text{rate})^{\text{nper}} - 1) = 0 \]

or, when \( \text{rate} = 0 \):

\[ \text{fv} + \text{pv} + \text{pmt} \times \text{nper} = 0 \]

for \( \text{pv} \), which is then returned.

References
[WRW]

Examples
What is the present value (e.g., the initial investment) of an investment that needs to total $15692.93 after 10 years of saving $100 every month? Assume the interest rate is 5% (annually) compounded monthly.

```python
>>> np.pv(0.05/12, 10*12, -100, 15692.93)
-100.00067131625819
```

By convention, the negative sign represents cash flow out (i.e., money not available today). Thus, to end up with $15,692.93 in 10 years saving $100 a month at 5% annual interest, one’s initial deposit should also be $100.

If any input is array_like, \( \text{pv} \) returns an array of equal shape. Let’s compare different interest rates in the example above:

```python
>>> a = np.array((0.05, 0.04, 0.03))/12
>>> np.pv(a, 10*12, -100, 15692.93)
array([-100.00067132, -649.26771385, -1273.78633713])
```

So, to end up with the same $15692.93 under the same $100 per month “savings plan,” for annual interest rates of 4% and 3%, one would need initial investments of $649.27 and $1273.79, respectively.

npv (rate, values)
Returns the NPV (Net Present Value) of a cash flow series.

Parameters
rate : scalar

3.16. Financial functions 645
The discount rate.

values : array_like, shape(M,)

The values of the time series of cash flows. The (fixed) time interval between cash flow “events” must be the same as that for which rate is given (i.e., if rate is per year, then precisely a year is understood to elapse between each cash flow event). By convention, investments or “deposits” are negative, income or “withdrawals” are positive; values must begin with the initial investment, thus values[0] will typically be negative.

Returns

out : float

The NPV of the input cash flow series values at the discount rate.

Notes

Returns the result of: [G61]

\[ \sum_{t=0}^{M} \frac{values_t}{(1 + rate)^t} \]

References

[G61]

Examples

```python
>>> np.npv(0.281, [-100, 39, 59, 55, 20])
-0.0066187288356340801
```

(Compare with the Example given for numpy.lib.financial.irr)

pmt : (rate, nper, pv, fv=0, when='end')

Compute the payment against loan principal plus interest.

Given:

- a present value, pv (e.g., an amount borrowed)
- a future value, fv (e.g., 0)
- an interest rate compounded once per period, of which there are
  - nper total
- and (optional) specification of whether payment is made at the beginning (when = {'begin', 1}) or the end (when = {'end', 0}) of each period

Return:

the (fixed) periodic payment.

Parameters

rate : array_like
  Rate of interest (per period)

nper : array_like
  Number of compounding periods
pv : array_like
    Present value

fv : array_like (optional)
    Future value (default = 0)

when : {{‘begin’, 1}, {‘end’, 0}}, {string, int}
    When payments are due (‘begin’ (1) or ‘end’ (0))

Returns
    out : ndarray
        Payment against loan plus interest. If all input is scalar, returns a scalar float. If any
        input is array_like, returns payment for each input element. If multiple inputs are array_like, they all must have the same shape.

Notes
The payment is computed by solving the equation:

\[ fv + pv \cdot (1 + rate)^{nper} + \]
\[ pmt \cdot \frac{(1 + rate \cdot when)}{rate} \cdot \frac{(1 + rate)^{nper} - 1}{rate} \] == 0

or, when rate == 0:

\[ fv + pv + pmt \cdot nper == 0 \]

for pmt.

Note that computing a monthly mortgage payment is only one use for this function. For example, pmt returns the
periodic deposit one must make to achieve a specified future balance given an initial deposit, a fixed, periodically
compounded interest rate, and the total number of periods.

References
[WRW]

Examples
What is the monthly payment needed to pay off a $200,000 loan in 15 years at an annual interest rate of 7.5%?

```python
>>> np.pmt(0.075/12, 12*15, 200000)
-1854.0247200054619
```

In order to pay-off (i.e., have a future-value of 0) the $200,000 obtained today, a monthly payment of $1,854.02
would be required. Note that this example illustrates usage of fv having a default value of 0.

ppmt (rate, per, nper, pv, fv=0.0, when=’end’)
Not implemented. Compute the payment against loan principal.

Parameters
rate : array_like
    Rate of interest (per period)

per : array_like, int
    Amount paid against the loan changes. The per is the period of interest.
**ipmt** *(rate, per, nper, pv, fv=0.0, when='end')*

Not implemented. Compute the payment portion for loan interest.

**Parameters**
- **rate**: scalar or array_like of shape(M, )
  - Rate of interest as decimal (not per cent) per period
- **per**: scalar or array_like of shape(M, )
  - Interest paid against the loan changes during the life or the loan. The *per* is the payment period to calculate the interest amount.
- **nper**: scalar or array_like of shape(M, )
  - Number of compounding periods
- **pv**: scalar or array_like of shape(M, )
  - Present value
- **fv**: scalar or array_like of shape(M, ), optional
  - Future value
- **when**: {{‘begin’, 1}, {‘end’, 0}}, {string, int}, optional
  - When payments are due (‘begin’ (1) or ‘end’ (0)). Defaults to {‘end’, 0}.

**Returns**
- **out**: ndarray
  - Interest portion of payment. If all input is scalar, returns a scalar float. If any input is array_like, returns interest payment for each input element. If multiple inputs are array_like, they all must have the same shape.

**See Also:**
- *ppmt*, *pmt*, *pv*

**Notes**
The total payment is made up of payment against principal plus interest.

\[ \text{pmt} = \text{ppmt} + \text{ipmt} \]
**irr** *(values)*

Return the Internal Rate of Return (IRR).

This is the “average” periodically compounded rate of return that gives a net present value of 0.0; for a more complete explanation, see Notes below.

**Parameters**

- **values**: array_like, shape(N,)

  Input cash flows per time period. By convention, net “deposits” are negative and net “withdrawals” are positive. Thus, for example, at least the first element of *values*, which represents the initial investment, will typically be negative.

**Returns**

- **out**: float

  Internal Rate of Return for periodic input values.

**Notes**

The IRR is perhaps best understood through an example (illustrated using np.irr in the Examples section below). Suppose one invests 100 units and then makes the following withdrawals at regular (fixed) intervals: 39, 59, 55, 20. Assuming the ending value is 0, one’s 100 unit investment yields 173 units; however, due to the combination of compounding and the periodic withdrawals, the “average” rate of return is neither simply 0.73/4 nor (1.73)^0.25-1. Rather, it is the solution (for *r*) of the equation:

\[ -100 + \frac{39}{1 + r} + \frac{59}{(1 + r)^2} + \frac{55}{(1 + r)^3} + \frac{20}{(1 + r)^4} = 0 \]

In general, for *values* = \([v_0, v_1, ... v_M]\), irr is the solution of the equation: [G41]

\[ \sum_{t=0}^{M} \frac{v_t}{(1 + irr)^t} = 0 \]

**References**

[G41]

**Examples**

```python
>>> np.irr([-100, 39, 59, 55, 20])
0.2809484211599611
```

(Compare with the Example given for numpy.lib.financial.npv)

**mirr** *(values, finance_rate, reinvest_rate)*

Modified internal rate of return.

**Parameters**

- **values**: array_like

  Cash flows (must contain at least one positive and one negative value) or nan is returned. The first value is considered a sunk cost at time zero.

- **finance_rate**: scalar

  Interest rate paid on the cash flows

- **reinvest_rate**: scalar

**3.16. Financial functions**
Interest rate received on the cash flows upon reinvestment

Returns

out : float

Modified internal rate of return

nper (rate, pmt, pv, fv=0, when='end')
Compute the number of periods.

Parameters

rate : array_like
    Rate of interest (per period)

pmt : array_like
    Payment

pv : array_like
    Present value

cfv : array_like, optional
    Future value

when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
    When payments are due ('begin' (1) or 'end' (0))

Notes

The number of periods nper is computed by solving the equation:

\[ fv + pv \times (1+rate)^{nper} + pmt \times \frac{(1+rate)^nper - 1}{rate \times (1+rate \times when)} = 0 \]

or, when \( rate = 0 \):

\[ fv + pv + pmt \times nper = 0 \]

Examples

If you only had $150 to spend as payment, how long would it take to pay-off a loan of $8,000 at 7% annual interest?

```python
>>> np.nper(0.07/12, -150, 8000)
64.073348770661852
```

So, over 64 months would be required to pay off the loan.

The same analysis could be done with several different interest rates and/or payments and/or total amounts to produce an entire table.

```python
>>> np.nper(*(np.ogrid[0.06/12:0.071/12:0.01/12, -200:-99:100, 6000:7001:1000]))
array([[ 32.58497782, 38.57048452],
       [ 71.51317802, 86.37179563],
       [ 33.07413144, 39.26244268],
       [ 74.06368256, 90.22989997]])
```

rate (nper, pmt, pv, fv; when='end', guess=0.10000000000000001, tol=9.9999999999999995e-07, maxiter=100)
Compute the rate of interest per period.
Parameters

**nper** : array_like
Number of compounding periods

**pmt** : array_like
Payment

**pv** : array_like
Present value

**fv** : array_like
Future value

**when** : {{‘begin’, 1}, {‘end’, 0}}, {string, int}, optional
When payments are due (‘begin’ (1) or ‘end’ (0))

**guess** : float, optional
Starting guess for solving the rate of interest

**tol** : float, optional
Required tolerance for the solution

**maxiter** : int, optional
Maximum iterations in finding the solution

Notes
The rate of interest is computed by iteratively solving the (non-linear) equation:

\[
fv + pv \times (1+rate)^{nper} + pmt \times (1+rate \times when) / rate \times ((1+rate)^{nper} - 1) = 0
\]

for rate.

References

3.17 Set routines

3.17.1 Making proper sets

```python
def unique(ar[, return_index, return_inverse])

Find the unique elements of an array.

unique(ar, return_index=False, return_inverse=False)

Find the unique elements of an array.
```

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.
Parameters

ar : array_like
    Input array. This will be flattened if it is not already 1-D.

return_index : bool, optional
    If True, also return the indices of ar that result in the unique array.

return_inverse : bool, optional
    If True, also return the indices of the unique array that can be used to reconstruct ar.

Returns

unique : ndarray
    The sorted unique values.

unique_indices : ndarray, optional
    The indices of the unique values in the (flattened) original array. Only provided if return_index is True.

unique_inverse : ndarray, optional
    The indices to reconstruct the (flattened) original array from the unique array. Only provided if return_inverse is True.

See Also:

numpy.lib.arraysetops
Module with a number of other functions for performing set operations on arrays.

Examples

```python
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
```

Return the indices of the original array that give the unique values:

```python
>>> a = np.array([['a', 'b', 'b', 'c', 'a']])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'],
      dtype='|S1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'],
      dtype='|S1')
```

Reconstruct the input array from the unique values:

```python
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
```
3.17.2 Boolean operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in1d</code></td>
<td>Test whether each element of a 1D array is also present in a second array.</td>
</tr>
<tr>
<td><code>intersect1d</code></td>
<td>Find the intersection of two arrays.</td>
</tr>
<tr>
<td><code>setdiff1d</code></td>
<td>Find the set difference of two arrays.</td>
</tr>
<tr>
<td><code>setxor1d</code></td>
<td>Find the set exclusive-or of two arrays.</td>
</tr>
<tr>
<td><code>union1d</code></td>
<td>Find the union of two arrays.</td>
</tr>
</tbody>
</table>

`in1d(ar1, ar2, assume_unique=False)`

Test whether each element of a 1D array is also present in a second array.

Returns a boolean array the same length as `ar1` that is True where an element of `ar1` is in `ar2` and False otherwise.

**Parameters**

- `ar1` : array_like, shape (M,)
  
  Input array.

- `ar2` : array_like
  
  The values against which to test each value of `ar1`.

- `assume_unique` : bool, optional
  
  If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

- `mask` : ndarray of bools, shape(M,)
  
  The values `ar1[mask]` are in `ar2`.

**See Also:**

`numpy.lib.arraysetops`

Module with a number of other functions for performing set operations on arrays.

**Notes**

`in1d` can be considered as an element-wise function version of the python keyword `in`, for 1D sequences. `in1d(a, b)` is roughly equivalent to `np.array([item in b for item in a])`. New in version 1.4.0.

**Examples**

```python
>>> test = np.array([0, 1, 2, 5, 0])
>>> states = [0, 2]
>>> mask = np.in1d(test, states)
>>> mask
array([ True,  False,  True,  False,  True], dtype=bool)
>>> test[mask]
array([0, 2, 0])
```
**intersect1d** *(ar1, ar2, assume_unique=False)*

Find the intersection of two arrays.

Return the sorted, unique values that are in both of the input arrays.

**Parameters**

* ar1, ar2 : array_like
  Input arrays.

* assume_unique : bool
  If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

* out : ndarray
  Sorted 1D array of common and unique elements.

**See Also:**

numpy.lib.arraysetops

Module with a number of other functions for performing set operations on arrays.

**Examples**

```python
>>> np.intersect1d([1, 3, 4, 3], [3, 1, 2, 1])
array([1, 3])
```

**setdiff1d** *(ar1, ar2, assume_unique=False)*

Find the set difference of two arrays.

Return the sorted, unique values in *ar1* that are not in *ar2*.

**Parameters**

* ar1 : array_like
  Input array.

* ar2 : array_like
  Input comparison array.

* assume_unique : bool
  If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

* difference : ndarray
  Sorted 1D array of values in *ar1* that are not in *ar2*.

**See Also:**

numpy.lib.arraysetops

Module with a number of other functions for performing set operations on arrays.

**Examples**
```python
>>> a = np.array([1, 2, 3, 2, 4, 1])
>>> b = np.array([3, 4, 5, 6])
>>> np.setdiff1d(a, b)
array([1, 2])

setxor1d (ar1, ar2, assume_unique=False)

Find the set exclusive-or of two arrays.

Return the sorted, unique values that are in only one (not both) of the input arrays.

Parameters

ar1, ar2 : array_like
    Input arrays.

assume_unique : bool
    If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

Returns

xor : ndarray
    Sorted 1D array of unique values that are in only one of the input arrays.

Examples

>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 3, 5, 7, 5])
>>> np.setxor1d(a, b)
array([1, 4, 5, 7])
```

union1d (ar1, ar2)

Find the union of two arrays.

Return the unique, sorted array of values that are in either of the two input arrays.

Parameters

ar1, ar2 : array_like
    Input arrays. They are flattened if they are not already 1D.

Returns

union : ndarray
    Unique, sorted union of the input arrays.

See Also:

numpy.lib.arraysetops
    Module with a number of other functions for performing set operations on arrays.

Examples

```python
>>> np.union1d([-1, 0, 1], [-2, 0, 2])
array([-2, -1,  0,  1,  2])
```
3.18 Window functions

3.18.1 Various windows

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bartlett(M)</td>
<td>Return the Bartlett window.</td>
</tr>
<tr>
<td>blackman(M)</td>
<td>Return the Blackman window.</td>
</tr>
<tr>
<td>hamming(M)</td>
<td>Return the Hamming window.</td>
</tr>
<tr>
<td>hanning(M)</td>
<td>Return the Hanning window.</td>
</tr>
<tr>
<td>kaiser(M, beta)</td>
<td>Return the Kaiser window.</td>
</tr>
</tbody>
</table>

**bartlett(M)**

Return the Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

**Parameters**

- M : int
  
  Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

- out : array
  
  The triangular window, normalized to one (the value one appears only if the number of samples is odd), with the first and last samples equal to zero.

**See Also:**

blackman, hamming, hanning, kaiser

**Notes**

The Bartlett window is defined as

\[
 w(n) = \frac{2}{M - 1} \left( \frac{M - 1}{2} - \left| n - \frac{M - 1}{2} \right| \right)
\]

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. The fourier transform of the Bartlett is the product of two sinc functions. Note the excellent discussion in Kanasewich.

**References**

[R18], [R19], [R20], [R21], [R22]

**Examples**

```python
>>> np.bartlett(12)
array([ 0. , 0.18181818, 0.36363636, 0.54545455, 0.72727273, 0.90909091, 0.90909091, 0.72727273, 0.54545455, 0.36363636, 0.18181818, 0. ])
```

Plot the window and its frequency response (requires SciPy and matplotlib):
```python
from numpy import clip, log10, array, bartlett, linspace
from numpy.fft import fft, fftshift
import matplotlib.pyplot as plt

window = bartlett(51)
plt.plot(window)
plt.title("Bartlett window")
plt.ylabel("Amplitude")
plt.xlabel("Sample")
plt.show()

plt.figure()
A = fft(window, 2048) / 25.5
mag = abs(fftshift(A))
freq = linspace(-0.5, 0.5, len(A))
response = 20*log10(mag)
response = clip(response, -100, 100)
plt.plot(freq, response)
plt.title("Frequency response of Blackman window")
plt.ylabel("Magnitude \[dB\]")
plt.xlabel("Normalized frequency \[cycles per sample\]"
plt.axis('tight')
plt.show()
```

3.18. Window functions
The Blackman window is a taper formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a Kaiser window.

**Parameters**

- `M` : int
  
  Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

- `out` : array
  
  The window, normalized to one (the value one appears only if the number of samples is odd).

**See Also:**

- `bartlett`, `hamming`, `hanning`, `kaiser`

**Notes**

The Blackman window is defined as

\[ w(n) = 0.42 - 0.5 \cos(2\pi n/M) + 0.08 \cos(4\pi n/M) \]

Most references to the Blackman window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. It is known as a “near optimal” tapering function, almost as good (by some measures) as the Kaiser window.

**References**

- [R24], [R25], [R26]
Examples

```python
>>> from numpy import blackman
>>> blackman(12)
array([-1.38777878e-17, 3.26064346e-02, 1.59903635e-01,
       4.14397981e-01, 7.36045180e-01, 9.67046769e-01,
       9.67046769e-01, 7.36045180e-01, 4.14397981e-01,
       1.59903635e-01, 3.26064346e-02, -1.38777878e-17])
```

Plot the window and the frequency response:

```python
>>> from numpy import clip, log10, array, bartlett, linspace
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt

>>> window = blackman(51)
>>> plt.plot(window)
>>> plt.title("Blackman window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.show()

>>> plt.figure()
>>> A = fft(window, 2048) / 25.5
>>> mag = abs(fftshift(A))
>>> freq = linspace(-0.5,0.5,len(A))
>>> response = 20*log10(mag)
>>> response = clip(response,-100,100)
>>> plt.plot(freq, response)
>>> plt.title("Frequency response of Blackman window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight')
>>> plt.show()
```

![Blackman window plot](image)
hamming($M$)

Return the Hamming window.

The Hamming window is a taper formed by using a weighted cosine.

**Parameters**

$M$ : int

Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

out : ndarray

The window, normalized to one (the value one appears only if the number of samples is odd).

**See Also:**

bartlett, blackman, hanning, kaiser

**Notes**

The Hamming window is defined as

\[ w(n) = 0.54 + 0.46 \cos\left(\frac{2\pi n}{M - 1}\right) \quad 0 \leq n \leq M - 1 \]

The Hamming was named for R. W. Hamming, an associate of J. W. Tukey and is described in Blackman and Tukey. It was recommended for smoothing the truncated autocovariance function in the time domain. Most references to the Hamming window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

**References**

[R30], [R31], [R32], [R33]
Examples

```python
>>> np.hamming(12)
array([ 0.08 , 0.15302337, 0.34890909, 0.60546483, 0.84123594,
       0.98136677, 0.98136677, 0.84123594, 0.60546483, 0.34890909,
       0.15302337, 0.08  ])
```

Plot the window and the frequency response:

```python
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt

>>> window = np.hamming(51)
>>> plt.plot(window)
>>> plt.title("Hamming window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.show()

>>> plt.figure()
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
>>> plt.title("Frequency response of Hamming window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight')
>>> plt.show()
```

3.18. Window functions
hanning\((M)\)

Return the Hanning window.

The Hanning window is a taper formed by using a weighted cosine.

**Parameters**

- \(M : \text{int}\)
  
  Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

- \(\text{out} : \text{ndarray, shape}(M,)\)
  
  The window, normalized to one (the value one appears only if \(M\) is odd).

**See Also:**

- bartlett, blackman, hamming, kaiser

**Notes**

The Hanning window is defined as

\[
   w(n) = 0.5 - 0.5 \cos \left( \frac{2\pi n}{M - 1} \right) \quad 0 \leq n \leq M - 1
\]

The Hanning was named for Julius van Hann, an Austrian meterologist. It is also known as the Cosine Bell. Some authors prefer that it be called a Hann window, to help avoid confusion with the very similar Hamming window.

Most references to the Hanning window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

**References**

- [R34], [R35], [R36], [R37]
Examples

```python
>>> from numpy import hanning
>>> hanning(12)
array([ 0.        , 0.07937323, 0.29229249, 0.57115742, 0.82743037,
       0.97974649, 0.97974649, 0.82743037, 0.57115742, 0.29229249,
       0.07937323, 0.        ])
```

Plot the window and its frequency response:

```python
>>> from numpy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt

>>> window = np.hanning(51)
>>> plt.plot(window)
>>> plt.title("Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.show()

>>> plt.figure()
>>> A = fft(window, 2048) / 25.5
>>> mag = abs(fftshift(A))
>>> freq = np.linspace(-0.5,0.5,len(A))
>>> response = 20*np.log10(mag)
>>> response = np.clip(response,-100,100)
>>> plt.plot(freq, response)
>>> plt.title("Frequency response of the Hann window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight')
>>> plt.show()
```

3.18. Window functions
kaiser$(M, \beta)$

Return the Kaiser window.

The Kaiser window is a taper formed by using a Bessel function.

**Parameters**

- $M$ : int
  - Number of points in the output window. If zero or less, an empty array is returned.

- $\beta$ : float
  - Shape parameter for window.

**Returns**

- out : array
  - The window, normalized to one (the value one appears only if the number of samples is odd).

**See Also:**

bartlett, blackman, hamming, hanning

**Notes**

The Kaiser window is defined as

$$w(n) = I_0 \left( \beta \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta)$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2},$$

where $I_0$ is the modified zeroth-order Bessel function.

The Kaiser was named for Jim Kaiser, who discovered a simple approximation to the DPSS window based on Bessel functions. The Kaiser window is a very good approximation to the Digital Prolate Spheroidal Sequence,
or Slepian window, which is the transform which maximizes the energy in the main lobe of the window relative to total energy.

The Kaiser can approximate many other windows by varying the beta parameter.

<table>
<thead>
<tr>
<th>beta</th>
<th>Window shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Rectangular</td>
</tr>
<tr>
<td>5</td>
<td>Similar to a Hamming</td>
</tr>
<tr>
<td>6</td>
<td>Similar to a Hanning</td>
</tr>
<tr>
<td>8.6</td>
<td>Similar to a Blackman</td>
</tr>
</tbody>
</table>

A beta value of 14 is probably a good starting point. Note that as beta gets large, the window narrows, and so the number of samples needs to be large enough to sample the increasingly narrow spike, otherwise nans will get returned.

Most references to the Kaiser window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R43], [R44], [R45]

Examples

```python
>>> from numpy import kaiser
>>> kaiser(12, 14)
array([ 7.72686684e-06, 3.46009194e-03, 4.65200189e-02,
       2.29737120e-01, 5.99885316e-01, 9.45674898e-01,
       9.45674898e-01, 5.99885316e-01, 2.29737120e-01,
       4.65200189e-02, 3.46009194e-03, 7.72686684e-06])
```

Plot the window and the frequency response:

```python
>>> from numpy import clip, log10, array, kaiser, linspace
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt

>>> window = kaiser(51, 14)
>>> plt.plot(window)
>>> plt.title("Kaiser window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.show()

>>> A = fft(window, 2048) / 25.5
>>> mag = abs(fftshift(A))
>>> freq = linspace(-0.5,0.5,len(A))
>>> response = 20*log10(mag)
>>> response = clip(response,-100,100)
>>> plt.plot(freq, response)
>>> plt.title("Frequency response of Kaiser window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight')
>>> plt.show()
```
3.19 Floating point error handling

3.19.1 Setting and getting error handling

```
seterr([all, divide, over, ...])  Set how floating-point errors are handled.
geterr()                         Get the current way of handling floating-point errors.
seterrcall(func)                 Set the floating-point error callback function or log object.
geterrcall()                     Return the current callback function used on floating-point errors.
errstate(**kwargs)              Context manager for floating-point error handling.
```

```
seterr(all=None, divide=None, over=None, under=None, invalid=None)
       Set how floating-point errors are handled.

       Note that operations on integer scalar types (such as int16) are handled like floating point, and are affected by these settings.
```
Parameters
   all : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
       Set treatment for all types of floating-point errors at once:
       • ignore: Take no action when the exception occurs.
       • warn: Print a RuntimeWarning (via the Python warnings module).
       • raise: Raise a FloatingPointError.
       • call: Call a function specified using the seterrcall function.
       • print: Print a warning directly to stdout.
       • log: Record error in a Log object specified by seterrcall.
       The default is not to change the current behavior.
   divide : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
       Treatment for division by zero.
   over : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
       Treatment for floating-point overflow.
   under : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
       Treatment for floating-point underflow.
   invalid : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
       Treatment for invalid floating-point operation.

Returns
   old_settings : dict
       Dictionary containing the old settings.

See Also:

   seterrcall
       Set a callback function for the 'call' mode.

   geterr, geterrcall

Notes
The floating-point exceptions are defined in the IEEE 754 standard [1]:
   • Division by zero: infinite result obtained from finite numbers.
   • Overflow: result too large to be expressed.
   • Underflow: result so close to zero that some precision was lost.
   • Invalid operation: result is not an expressible number, typically indicates that a NaN was produced.

Examples

>>> np.seterr(over='raise')
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore',
 'under': 'ignore'}
>>> np.seterr(all='ignore')  # reset to default
{'over': 'raise', 'divide': 'warn', 'invalid': 'warn', 'under': 'warn'}
```python
>>> np.int16(32000) * np.int16(3)
30464
>>> old_settings = np.seterr(all='warn', over='raise')
>>> np.int16(32000) * np.int16(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: overflow encountered in short_scalars

>>> np.seterr(all='print')
{'over': 'print', 'divide': 'print', 'invalid': 'print', 'under': 'print'}
>>> np.int16(32000) * np.int16(3)
Warning: overflow encountered in short_scalars
30464

Calling `seterr` with no arguments resets treatment for all floating-point errors to the defaults.

```
There are two ways to capture floating-point error messages. The first is to set the error-handler to ‘call’, using \texttt{seterr}. Then, set the function to call using this function.

The second is to set the error-handler to ‘log’, using \texttt{seterr}. Floating-point errors then trigger a call to the ‘write’ method of the provided object.

\textbf{Parameters}

- \texttt{func} : callable \texttt{f(err, flag)} or object with write method

  Function to call upon floating-point errors (‘call’-mode) or object whose ‘write’ method is used to log such message (‘log’-mode).

  The call function takes two arguments. The first is the type of error (one of “divide”, “over”, “under”, or “invalid”), and the second is the status flag. The flag is a byte, whose least-significant bits indicate the status:

  \[
  [0 \ 0 \ 0 \ 0 \ \text{invalid} \ \over \ \text{under} \ \text{invalid}]
  \]

  In other words, \texttt{flags = divide + 2*over + 4*under + 8*invalid}.

  If an object is provided, its write method should take one argument, a string.

\textbf{Returns}

- \texttt{h} : callable, log instance or None

  The old error handler.

\textbf{See Also:}

\texttt{seterr, geterr, geterrcall}

\textbf{Examples}

\textbf{Callback upon error:}

\begin{verbatim}
>>> def err_handler(type, flag):
...   print "Floating point error (%s), with flag %s" % (type, flag)
...

>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')

>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([ Inf,  Inf,  Inf])

>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)

\end{verbatim}

\textbf{Log error message:}

\begin{verbatim}
>>> class Log(object):
...   def write(self, msg):
...     print "LOG: %s" % msg
...

\end{verbatim}
```python
>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')

>>> np.array([1, 2, 3]) / 0.0
LOG: Warning: divide by zero encountered in divide

>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)
```

### geterrcall()

Return the current callback function used on floating-point errors.

When the error handling for a floating-point error (one of “divide”, “over”, “under”, or “invalid”) is set to ‘call’ or ‘log’, the function that is called or the log instance that is written to is returned by `geterrcall`. This function or log instance has been set with `seterrcall`.

Returns:
- `errobj`: callable, log instance or None

The current error handler. If no handler was set through `seterrcall`, None is returned.

**See Also:**
- `seterrcall`, `seterr`, `geterr`

**Notes**

For complete documentation of the types of floating-point exceptions and treatment options, see `seterr`.

**Examples**

```python
>>> np.geterrcall()  # we did not yet set a handler, returns None
```

```python
>>> oldsettings = np.seterr(all='call')
>>> def err_handler(type, flag):
...     print "Floating point error (%s), with flag %s" % (type, flag)
>>> oldhandler = np.seterrcall(err_handler)
>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([ Inf, Inf, Inf])

>>> cur_handler = np.geterrcall()
>>> cur_handler is err_handler
True
```

### class `errstate(**kwargs)`

Context manager for floating-point error handling.

Using an instance of `errstate` as a context manager allows statements in that context to execute with a known error handling behavior. Upon entering the context the error handling is set with `seterr` and `seterrcall`, and upon exiting it is reset to what it was before.

**Parameters**
- `kwargs`: {divide, over, under, invalid}
Keyword arguments. The valid keywords are the possible floating-point exceptions. Each keyword should have a string value that defines the treatment for the particular error. Possible values are {'ignore', 'warn', 'raise', 'call', 'print', 'log'}.

See Also:

seterr, geterr, seterrcall, geterrcall

Notes

The `with` statement was introduced in Python 2.5, and can only be used there by importing it: `from __future__ import with_statement`. In earlier Python versions the `with` statement is not available.

For complete documentation of the types of floating-point exceptions and treatment options, see `seterr`.

Examples

```python
>>> from __future__ import with_statement  # use 'with' in Python 2.5
>>> olderr = np.seterr(all='ignore')  # Set error handling to known state.

>>> np.arange(3) / 0.
array([ NaN, Inf, Inf])
>>> with np.errstate(divide='warn'):
...     np.arange(3) / 0.
...__main__:2: RuntimeWarning: divide by zero encountered in divide
     array([ NaN, Inf, Inf])

>>> np.sqrt(-1)
nan
>>> with np.errstate(invalid='raise'):
...     np.sqrt(-1)
...Traceback (most recent call last):
 ... File "<stdin>", line 2, in <module>
 FloatingPointError: invalid value encountered in sqrt

Outside the context the error handling behavior has not changed:

```python
>>> np.geterr()
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}
```

3.19.2 Internal functions

| seterrobj (errobj) | Set the object that defines floating-point error handling. |
| geterrobj () | Return the current object that defines floating-point error handling. |

`seterrobj`()

Set the object that defines floating-point error handling.

The error object contains all information that defines the error handling behavior in Numpy. `seterrobj` is used internally by the other functions that set error handling behavior (`seterr`, `seterrcall`).

Parameters

- `errobj` : list
The error object, a list containing three elements: [internal numpy buffer size, error mask, error callback function].

The error mask is a single integer that holds the treatment information on all four floating point errors. The information for each error type is contained in three bits of the integer. If we print it in base 8, we can see what treatment is set for “invalid”, “under”, “over”, and “divide” (in that order). The printed string can be interpreted with:

- 0: ‘ignore’
- 1: ‘warn’
- 2: ‘raise’
- 3: ‘call’
- 4: ‘print’
- 5: ‘log’

**See Also:**

geterrobj, seterr, geterr, seterrcall, geterrcall, getbufsize, setbufsize

**Notes**

For complete documentation of the types of floating-point exceptions and treatment options, see seterr.

**Examples**

```python
>>> old_errobj = np.geterrobj()  # first get the defaults
>>> old_errobj
[10000, 0, None]

>>> def err_handler(type, flag):
...     print "Floating point error (%s), with flag %s" % (type, flag)
...
>>> new_errobj = [20000, 12, err_handler]
>>> np.seterrobj(new_errobj)
>>> np.base_repr(12, 8)  # int for divide=4 ('print') and over=1 ('warn')
'14'
>>> np.geterr()
{'over': 'warn', 'divide': 'print', 'invalid': 'ignore', 'under': 'ignore'}
>>> np.geterrcall() is err_handler
True
```

**geterrobj()**

Return the current object that defines floating-point error handling.

The error object contains all information that defines the error handling behavior in Numpy. geterrobj is used internally by the other functions that get and set error handling behavior (geterr, seterr, geterrcall, seterrcall).

Returns

errobj : list

The error object, a list containing three elements: [internal numpy buffer size, error mask, error callback function].

The error mask is a single integer that holds the treatment information on all four floating point errors. The information for each error type is contained in three bits of the integer. If we print it in base 8, we can see what treatment is set for “invalid”, “under”, “over”, and “divide” (in that order). The printed string can be interpreted with
• 0: ‘ignore’
• 1: ‘warn’
• 2: ‘raise’
• 3: ‘call’
• 4: ‘print’
• 5: ‘log’

See Also:
seterrobj, seterr, geterr, seterrcall, geterrcall, getbufsize, setbufsize

Notes
For complete documentation of the types of floating-point exceptions and treatment options, see seterr.

Examples

```python
>>> np.geterrobj() # first get the defaults
[10000, 0, None]

>>> def err_handler(type, flag):
...     print "Floating point error (%s), with flag %s" % (type, flag)
...

>>> old_bufsize = np.setbufsize(20000)
>>> old_err = np.seterr(divide='raise')
>>> old_handler = np.seterrcall(err_handler)
>>> np.geterrobj()
[20000, 2, <function err_handler at 0x91dcaac>]

>>> old_err = np.seterr(all='ignore')
>>> np.base_repr(np.geterrobj()[1], 8)
'0'

>>> old_err = np.seterr(divide='warn', over='log', under='call', invalid='print')
>>> np.base_repr(np.geterrobj()[1], 8)
'4351'
```

3.20 Masked array operations

3.20.1 Constants

<table>
<thead>
<tr>
<th>ma.MaskType</th>
<th>Numpy’s Boolean type. Character code: ?. Alias: bool8</th>
</tr>
</thead>
</table>

MaskType
alias of bool_
3.20.2 Creation

From existing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.masked_array</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.array(data[, dtype, copy, order, ...])</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.copy()</code></td>
<td>Copy a.copy(order='C')</td>
</tr>
<tr>
<td><code>ma.frombuffer(buffer[, dtype, count, offset])</code></td>
<td>Interpret a buffer as a 1-dimensional array.</td>
</tr>
<tr>
<td><code>ma.fromfunction(function, shape, **kwargs)</code></td>
<td>Construct an array by executing a function over each coordinate.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.copy([order])</code></td>
<td>Return a copy of the array.</td>
</tr>
</tbody>
</table>

**masked_array**

alias of `MaskedArray`

**array**(data, dtype=None, copy=False, order=False, mask=False, fill_value=None, keep_mask=True, hard_mask=False, shrink=True, subok=True, ndmin=0)

An array class with possibly masked values.

Masked values of True exclude the corresponding element from any computation.

Construction:

```python
x = MaskedArray(data, mask=nomask, dtype=None, copy=True, fill_value=None, keep_mask=True, hard_mask=False, shrink=True)
```

**Parameters**

- **data**: array_like
  
  Input data.

- **mask**: sequence, optional
  
  Mask. Must be convertible to an array of booleans with the same shape as `data`. True indicates a masked (i.e. invalid) data.

- **dtype**: dtype, optional
  
  Data type of the output. If `dtype` is None, the type of the data argument (data.dtype) is used. If `dtype` is not None and different from data.dtype, a copy is performed.

- **copy**: bool, optional
  
  Whether to copy the input data (True), or to use a reference instead. Default is False.

- **subok**: bool, optional
  
  Whether to return a subclass of `MaskedArray` if possible (True) or a plain `MaskedArray`. Default is True.

- **ndmin**: int, optional
  
  Minimum number of dimensions. Default is 0.

- **fill_value**: scalar, optional
  
  Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

- **keep_mask**: bool, optional
  
  Whether to combine `mask` with the mask of the input data, if any (True), or to use only `mask` for the output (False). Default is True.
**hard_mask** : bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be un-masked. Default is False.

**shrink** : bool, optional

Whether to force compression of an empty mask. Default is True.

**copy**

```python
copy a.copy(order='C')
```

Return a copy of the array.

**Parameters**

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

By default, the result is stored in C-contiguous (row-major) order in memory. If *order* is *F*, the result has ‘Fortran’ (column-major) order. If *order* is ‘A’ (‘Any’), then the result has the same order as the input.

**Examples**

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

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

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

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

**frombuffer**

Interpret a buffer as a 1-dimensional array.

**Parameters**

**buffer** :

An object that exposes the buffer interface.

**dtype** : data-type, optional

Data type of the returned array.

**count** : int, optional

Number of items to read. -1 means all data in the buffer.

**offset** : int, optional

Start reading the buffer from this offset.
Notes
If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```python
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```python
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

`fromfunction()`
Construct an array by executing a function over each coordinate.

The resulting array therefore has a value $fn(x, y, z)$ at coordinate $(x, y, z)$.

Parameters

- `function`: callable
  The function is called with N parameters, each of which represents the coordinates of the array varying along a specific axis. For example, if `shape` were $(2, 2)$, then the parameters would be two arrays, $[[0, 0], [1, 1]]$ and $[[0, 1], [0, 1]]$. `function` must be capable of operating on arrays, and should return a scalar value.

- `shape`: (N,) tuple of ints
  Shape of the output array, which also determines the shape of the coordinate arrays passed to `function`.

- `dtype`: data-type, optional
  Data-type of the coordinate arrays passed to `function`. By default, `dtype` is float.

Returns

- `out`: any
  The result of the call to `function` is passed back directly. Therefore the type and shape of `out` is completely determined by `function`.

See Also:
`indices`, `meshgrid`

Notes
Keywords other than `shape` and `dtype` are passed to `function`.

Examples

```python
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)
```
```python
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

**copy()**

Return a copy of the array.

**Parameters**

- `order`: {'C', 'F', 'A'}, optional

By default, the result is stored in C-contiguous (row-major) order in memory. If `order` is `F`, the result has ‘Fortran’ (column-major) order. If order is ‘A’ (‘Any’), then the result has the same order as the input.

**Examples**

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

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

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

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

**Ones and zeros**

- `ma.empty(shape[, dtype, order])`  
  Return a new array of given shape and type, without initializing entries.
  
- `ma.empty_like(a)`  
  Return a new array with the same shape and type as a given array.
  
- `ma.masked_all(shape[, dtype])`  
  Empty masked array with all elements masked.
  
- `ma.masked_all_like(arr)`  
  Empty masked array with the properties of an existing array.
  
- `ma.ones(shape[, dtype, order])`  
  Return a new array of given shape and type, filled with ones.
  
- `ma.zeros(shape[, dtype, order])`  
  Return a new array of given shape and type, filled with zeros.

**empty()**

Return a new array of given shape and type, without initializing entries.

**Parameters**

- `shape`: int or tuple of int  
  Shape of the empty array

- `dtype`: data-type, optional  
  Desired output data-type.
order: {‘C’, ‘F’}, optional

Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

See Also:
empty_like, zeros, ones

Notes
empty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples
>>> np.empty([2, 2])
array([[ -9.74499359e+001,  6.69583040e-309],
       [ 2.13182611e-314,  3.06959433e-309]])

>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],
       [ 496041986, 19249760]])

empty_like()
Return a new array with the same shape and type as a given array.

Parameters
a : array_like

The shape and data-type of a define the parameters of the returned array.

Returns
out : ndarray

Array of random data with the same shape and type as a.

See Also:
ones_like
Return an array of ones with shape and type of input.

zeros_like
Return an array of zeros with shape and type of input.

empty
Return a new uninitialized array.

ones
Return a new array setting values to one.

zeros
Return a new array setting values to zero.

Notes
This function does not initialize the returned array; to do that use zeros_like or ones_like instead. It may be marginally faster than the functions that do set the array values.
Examples

```python
>>> a = ([1, 2, 3], [4, 5, 6])  # a is array-like
>>> np.empty_like(a)
array([[-1073741821, -1073741821, 3],  #random
       [0, 0, -1073741821]])
>>> a = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> np.empty_like(a)
array([[ -2.00000715e+000, 1.48219694e-323, -2.00000572e+000],  #random
       [4.38791518e-305, -2.00000715e+000, 4.17269252e-309]])
```

**masked_all** *(shape, dtype=<type 'float'>)*

Empty masked array with all elements masked.

Return an empty masked array of the given shape and dtype, where all the data are masked.

**Parameters**

- **shape**: tuple
  Shape of the required MaskedArray.

- **dtype**: dtype, optional
  Data type of the output.

**Returns**

- **a**: MaskedArray
  A masked array with all data masked.

**See Also:**

**masked_all_like**

Empty masked array modelled on an existing array.

**Examples**

```python
>>> import numpy.ma as ma
>>> ma.masked_all((3, 3))
mixed_array(data =
        [[-- -- --],
         [-- -- --],
         [-- -- --]],
        mask =
        [[ True  True  True],
         [ True  True  True],
         [ True  True  True]],
        fill_value=1e+20)
```

The **dtype** parameter defines the underlying data type.

```python
>>> a = ma.masked_all((3, 3))
>>> a.dtype
dtype('float64')
```

**masked_all_like** *(arr)*

Empty masked array with the properties of an existing array.
Return an empty masked array of the same shape and dtype as the array `arr`, where all the data are masked.

**Parameters**

- `arr` : ndarray
  
  An array describing the shape and dtype of the required MaskedArray.

**Returns**

- `a` : MaskedArray
  
  A masked array with all data masked.

**Raises**

- `AttributeError`
  
  If `arr` doesn’t have a shape attribute (i.e. not an ndarray)

**See Also:**

- `masked_all`
  
  Empty masked array with all elements masked.

**Examples**

```python
>>> import numpy.ma as ma
>>> arr = np.zeros((2, 3), dtype=np.float32)
>>> arr
array([[ 0., 0., 0.],
       [ 0., 0., 0.]], dtype=float32)

>>> ma.masked_all_like(arr)
masked_array(data = [-- -- --],
             mask = [[ True True True],
                     [ True True True]],
             fill_value=1e+20)
```

The dtype of the masked array matches the dtype of `arr`.

```python
>>> arr.dtype
dtype('float32')
>>> ma.masked_all_like(arr).dtype
dtype('float32')
```

**ones**

Return a new array of given shape and type, filled with ones.

**See Also:**

- `zeros`

**Examples**

```python
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
```
zeros()  
Return a new array of given shape and type, filled with zeros.

Parameters

- **shape**: int or sequence of ints
  Shape of the new array, e.g., (2, 3) or 2.
- **dtype**: data-type, optional
  The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.
- **order**: {'C', 'F'}, optional
  Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns

- **out**: ndarray
  Array of zeros with the given shape, dtype, and order.

See Also:

- **zeros_like**: Return an array of zeros with shape and type of input.
- **ones_like**: Return an array of ones with shape and type of input.
- **empty_like**: Return an empty array with shape and type of input.
- **ones**: Return a new array setting values to one.
- **empty**: Return a new uninitialized array.

Examples

```python
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])
```

```python
>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])
```
```python
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])

>>> s = (2, 2)
>>> np.zeros(s)
array([[ 0., 0.],
       [ 0., 0.]])

>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')])
# custom dtype
array([(0, 0), (0, 0)],
dtype=[('x', '<i4'), ('y', '<i4')])
```

3.20.3 Inspecting the array

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.all()</code></td>
<td>Check if all of the elements of a are true.</td>
</tr>
<tr>
<td><code>ma.any()</code></td>
<td>Check if any of the elements of a are true.</td>
</tr>
<tr>
<td><code>ma.count(a, axis)</code></td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td><code>ma.count_masked(arr, axis)</code></td>
<td>Count the number of masked elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.getmask(a)</code></td>
<td>Return the mask of a masked array, or nomask.</td>
</tr>
<tr>
<td><code>ma.getmaskarray(arr)</code></td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td><code>ma.getdata(a, subok)</code></td>
<td>Return the data of a masked array as an ndarray.</td>
</tr>
<tr>
<td><code>ma.getmaskarray()</code></td>
<td>Return the mask of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>ma.shape(obj)</code></td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td><code>ma.size(obj, axis)</code></td>
<td>Return the number of elements along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.data</code></td>
<td>Return the current data, as a view of the original underlying data.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.mask</code></td>
<td>Return the mask of the records. A record is masked when all the fields are masked.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.count()</code></td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.nonzero()</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>ma.shape(obj)</code></td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td><code>ma.size(obj, axis)</code></td>
<td>Return the number of elements along a given axis.</td>
</tr>
</tbody>
</table>

**all()**

Check if all of the elements of a are true.

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**

- **axis**: [None, integer]
  
  Axis to perform the operation over. If None, perform over flattened array.

- **out**: [None, array], optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.
See Also:

all  
equivalent function

Examples

```python
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

any ()
Check if any of the elements of a are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

Parameters
axis : {None, integer}
Axis to perform the operation over. If None, perform over flattened array and return a scalar.

out : {None, array}, optional
Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

any  
equivalent function

count (a, axis=None)
Count the non-masked elements of the array along the given axis.

Parameters
axis : int, optional
Axis along which to count the non-masked elements. If axis is None, all non-masked elements are counted.

Returns
result : int or ndarray
If axis is None, an integer count is returned. When axis is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:

count_masked  
Count masked elements in array or along a given axis.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
```
```python
>>> a
masked_array(data =
[[0 1 2]
[-- -- --]],
mask =
[[False False False]
[ True  True True]],
fill_value = 999999)
>>> a.count()
3

When the `axis` keyword is specified an array of appropriate size is returned.

```python
>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])
```

`count_masked(arr, axis=None)`
Count the number of masked elements along the given axis.

**Parameters**

- **arr**: array_like
  An array with (possibly) masked elements.

- **axis**: int, optional
  Axis along which to count. If None (default), a flattened version of the array is used.

**Returns**

- **count**: int, ndarray
  The total number of masked elements (axis=None) or the number of masked elements along each slice of the given axis.

**See Also:**

`MaskedArray.count`
Count non-masked elements.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(9).reshape((3,3))
>>> a = ma.array(a)
>>> a[1, 0] = ma.masked
>>> a[1, 2] = ma.masked
>>> a[2, 1] = ma.masked
>>> a
masked_array(data =
[[0 1 2]
[-- 4 --]
[6 -- 8]],
mask =
[[False False False]
[ True False True]
[False False False]],
fill_value=999999)
```
>>> ma.count_masked(a)
3

When the `axis` keyword is used an array is returned.

>>> ma.count_masked(a, axis=0)
array([1, 1, 1])

>>> ma.count_masked(a, axis=1)
array([0, 2, 1])

getmask(a)

Return the mask of a masked array, or nomask.

Return the mask of `a` as an ndarray if `a` is a `MaskedArray` and the mask is not `nomask`, else return `nomask`. To guarantee a full array of booleans of the same shape as `a`, use `getmaskarray`.

Parameters

- `a`: array_like
  Input `MaskedArray` for which the mask is required.

See Also:

gedata

Return the data of a masked array as an ndarray.

getmaskarray

Return the mask of a masked array, or full array of False.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False True]
[False False]], fill_value=999999)

>>> ma.getmask(a)
array([[False, True],
       [False, False]], dtype=bool)
```

Equivalently use the `MaskedArray mask` attribute.

```python
>>> a.mask
array([[False, True],
       [False, False]], dtype=bool)
```

Result when mask == `nomask`

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
[3 4]],
mask =
[[False False],
[False False]],
fill_value=999999)
```
```python
mask =
False,
fill_value=999999)
>>> ma.nomask
False
>>> ma.getmask(b) == ma.nomask
True
>>> b.mask == ma.nomask
True

getmaskarray (arr)

Return the mask of a masked array, or full boolean array of False.

Return the mask of arr as an ndarray if arr is a MaskedArray and the mask is not nomask, else return a full boolean array of False of the same shape as arr.

Parameters

arr : array_like
    Input MaskedArray for which the mask is required.

See Also:

getmask
    Return the mask of a masked array, or nomask.

ggetdata
    Return the data of a masked array as an ndarray.

Examples

>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[ [ 1 --]
 [ 3 4]],
mask =
[[False True]
[False False]],
fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
[False, False]], dtype=bool)

Result when mask == nomask

>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[ [ 1 2]
 [ 3 4]],
mask =
False,
fill_value=999999)
>>> >ma.getmaskarray(b)
array([[False, False],
[False, False]], dtype=bool)
```
getdata (a, subok=True)

Return the data of a masked array as an ndarray.

Return the data of a (if any) as an ndarray if a is a MaskedArray, else return a as a ndarray or subclass (depending on subok) if not.

Parameters

a : array_like

Input MaskedArray, alternatively a ndarray or a subclass thereof.

subok : bool

Whether to force the output to be a pure ndarray (False) or to return a subclass of ndarray if appropriate (True, default).

See Also:

getmask

Return the mask of a masked array, or nomask.

getmaskarray

Return the mask of a masked array, or full array of False.

Examples

>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False True]
[False False]],
fill_value=999999)
>>> ma.getdata(a)
array([[1, 2],
[3, 4]])

Equivalently use the MaskedArray data attribute.

>>> a.data
array([[1, 2],
[3, 4]])

nonzero()

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

a[a.nonzero()]

To group the indices by element, rather than dimension, use instead:

np.transpose(a.nonzero())

The result of this is always a 2d array, with a row for each non-zero element.
Parameters

None:

Returns

tuple_of_arrays : tuple

Indices of elements that are non-zero.

See Also:

numpy.nonzero
Function operating on ndarrays.

flatnonzero
Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero
Equivalent ndarray method.

Examples

```python
>>> import numpy as np
>>> x = np.eye(3)
>>> x
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = np.ma.masked
>>> x
masked_array(data =
[[1.0 0.0 0.0]
[0.0 -- 0.0]
[0.0 0.0 1.0]],
mask =
[[False False False]
[False True False]
[False False False]],
fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))

Indices can also be grouped by element.

>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array a, the condition a > 3 is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the a where the condition is true.
>>> a = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
masked_array(data =
[ [False False False]
[ True True True]
[ True True True]],
mask =
False,
fill_value=999999)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

The nonzero method of the condition array can also be called.

>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

shape (obj)
Return the shape of an array.

Parameters
a : array_like
Input array.

Returns
shape : tuple of ints
The elements of the shape tuple give the lengths of the corresponding array dimensions.

See Also:
alen

ndarray.shape
Equivalent array method.

Examples

>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()

>>> a = np.array([[1, 2], [3, 4]], dtype=[('x', 'i4'), ('y', 'i8')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)

size (obj, axis=None)
Return the number of elements along a given axis.

Parameters
a : array_like
Input data.

**axis**: `int`, optional

Axis along which the elements are counted. By default, give the total number of elements.

**Returns**

**element_count**: `int`

Number of elements along the specified axis.

**See Also:**

**shape**

dimensions of array

**ndarray.shape**

dimensions of array

**ndarray.size**

number of elements in array

**Examples**

```python
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.size(a)
6
>>> np.size(a,1)
3
>>> np.size(a,0)
2
```

data

Return the current data, as a view of the original underlying data.

**mask**

Mask

**recordmask**

Return the mask of the records. A record is masked when all the fields are masked.

**all** (`axis=None, out=None`)

Check if all of the elements of `a` are true.

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**

axis : `{None, integer}`

Axis to perform the operation over. If None, perform over flattened array.

out : `{None, array}`, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

**all**

equivalent function
Examples

```python
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

any (axis=None, out=None)
Check if any of the elements of `a` are true.
Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

Parameters

axis : {None, integer}
Axis to perform the operation over. If None, perform over flattened array and return a scalar.

out : {None, array}, optional
Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

any
equivalent function

count (axis=None)
Count the non-masked elements of the array along the given axis.

Parameters

axis : int, optional
Axis along which to count the non-masked elements. If `axis` is None, all non-masked elements are counted.

Returns

result : int or ndarray
If `axis` is None, an integer count is returned. When `axis` is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:

count_masked
Count masked elements in array or along a given axis.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
```

When the axis keyword is specified an array of appropriate size is returned.

```python
>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])
```

nonzero()  
Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```python
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use instead:

```python
np.transpose(a.nonzero())
```

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

None:

**Returns**

tuple_of_arrays : tuple

Indices of elements that are non-zero.

**See Also:**

- `numpy.nonzero`
  - Function operating on ndarrays.
- `flatnonzero`
  - Return indices that are non-zero in the flattened version of the input array.
- `ndarray.nonzero`
  - Equivalent ndarray method.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
    [[ 1.  0.  0.]
     [ 0.  1.  0.]
     [ 0.  0.  1.]],
    mask =
    [[False False False]
     [ True  True True]],
    fill_value = 999999)
>>> x.count()
3
```
```python
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

```python
>>> x[1, 1] = ma.masked
``` """ma""" \\
```python
>>> x
masked_array(data =
[[1.0 0.0 0.0]
[0.0 -- 0.0]
[0.0 0.0 1.0]],
mask =
[[False False False]
[False True False]
[False False False]],
fill_value=1e+20)
``` """ma""" \\
```python
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
``` """ma""" 

Indices can also be grouped by element.

```python
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
``` """ma""" 

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array \( a \), the condition \( a > 3 \) is a boolean array and since False is interpreted as 0, \( \text{ma}.\text{nonzero}(a > 3) \) yields the indices of the \( a \) where the condition is true.

```python
>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(data =
[[False False False]
 [ True True True]
 [ True True True]],
mask =
False,
fill_value=999999)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
``` """ma""" 

The nonzero method of the condition array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
``` """ma""" 

**shape** *(obj)*

Return the shape of an array.

**Parameters**

- `a` : array_like

  Input array.

**Returns**

- `shape` : tuple of ints

  The elements of the shape tuple give the lengths of the corresponding array dimensions.
See Also:

alen

**ndarray.shape**
Equivalent array method.

**Examples**

```python
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```python
>>> a = np.array([[1, 2], [3, 4]], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
```

**size**(obj, axis=None)
Return the number of elements along a given axis.

**Parameters**

- **a**: array_like
  Input data.
- **axis**: int, optional
  Axis along which the elements are counted. By default, give the total number of elements.

**Returns**

- **element_count**: int
  Number of elements along the specified axis.

See Also:

- **shape**: dimensions of array
- **ndarray.shape**: dimensions of array
- **ndarray.size**: number of elements in array

**Examples**

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.size(a)
6
>>> np.size(a, 1)
3
```
3.20.4 Manipulating a MaskedArray

Changing the shape

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.ravel(self)</td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td>ma.reshape(a, new_shape[, order])</td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>ma.resize(x, new_shape)</td>
<td>Return a new masked array with the specified size and shape.</td>
</tr>
<tr>
<td>ma.MaskedArray.flatten([order])</td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td>ma.MaskedArray.ravel(self)</td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td>ma.MaskedArray.reshape(self, *s, **kwargs)</td>
<td>Give a new shape to the array without changing its data.</td>
</tr>
<tr>
<td>ma.MaskedArray.resize(self, newshape[, refcheck, order])</td>
<td></td>
</tr>
</tbody>
</table>

**ravel()**

Returns a 1D version of self, as a view.

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
[7 -- 9]]
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

**reshape(a, new_shape, order='C')**

Returns an array containing the same data with a new shape.

Refer to `MaskedArray.reshape` for full documentation.

**See Also:**

- `MaskedArray.reshape`
- `numpy.resize`

**resize(x, new_shape)**

Return a new masked array with the specified size and shape.

This is the masked equivalent of the `numpy.resize` function. The new array is filled with repeated copies of x (in the order that the data are stored in memory). If x is masked, the new array will be masked, and the new mask will be a repetition of the old one.
See Also:

numpy.resize
   Equivalent function in the top level NumPy module.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.array([[1, 2], [3, 4]])
>>> a[0, 1] = ma.masked
>>> a
masked_array(data =
[1 --]
[3 4]),
   mask =
[[False  True]
[False False]],
   fill_value = 999999)
```
```
>>> np.resize(a, (3, 3))
array([[1, 2, 3],
[4, 1, 2],
[3, 4, 1]])
```
```
>>> ma.resize(a, (3, 3))
masked_array(data =
[1 -- 3]
[4 1 --]
[3 4 1]),
   mask =
[[False  True False]
[False False True]
[False False False]],
   fill_value = 999999)
```

A MaskedArray is always returned, regardless of the input type.

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> ma.resize(a, (3, 3))
masked_array(data =
[1 2 3]
[4 1 2]
[3 4 1]),
   mask =
False,
   fill_value = 999999)
```

flatten()

Return a copy of the array collapsed into one dimension.

Parameters

order : {'C', 'F'}, optional
   Whether to flatten in C (row-major) or Fortran (column-major) order. 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])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

**ravel()**
Returns a 1D version of self, as a view.

**Returns**

**MaskedArray**:

Output view is of shape `(self.size,)` (or `(np.ma.product(self.shape),)`).

**Examples**

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
[7 -- 9]]
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

**reshape (**s, **kwargs)**
Give a new shape to the array without changing its data.

Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

**Parameters**

- **shape** : int or tuple of ints
  The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

- **order** : {'C', 'F'}, optional
  Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

**Returns**

**reshaped_array** : array
A new view on the array.

See Also:

**reshape**
Equivalent function in the masked array module.
**numpy.ndarray.reshape**
Equivalent method on ndarray object.

**numpy.reshape**
Equivalent function in the NumPy module.

**Notes**
The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`.

**Examples**
```python
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
>>> print x
[-- 2]
[3 --]
>>> x = x.reshape((4,1))
>>> print x
[[-]
[2]
[3]
[--]
```

**resize** *(newshape, refcheck=True, order=False)*

*Warning:* This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the `numpy.ma.resize` function instead.

This method is difficult to implement safely and may be deprecated in future releases of NumPy.

**Modifying axes**

<table>
<thead>
<tr>
<th>swapaxes()</th>
<th>swapaxes(a.swapaxes(axis1, axis2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.swapaxes()</td>
<td>Permute the dimensions of an array.</td>
</tr>
<tr>
<td>ma.transpose(a[, axes])</td>
<td>Return a view of the array with <code>axis1</code> and <code>axis2</code> interchanged.</td>
</tr>
<tr>
<td>ma.MaskedArray.swapaxes(axis1, axis2)</td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td>ma.MaskedArray.transpose(*axes)</td>
<td></td>
</tr>
</tbody>
</table>

**See Also:**

**numpy.swapaxes**
equivalent function

**transpose(a, axes=None)**
Permute the dimensions of an array.

This function is exactly equivalent to `numpy.transpose`.

**See Also:**
**numpy.transpose**
Equivalent function in top-level NumPy module.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked

masked_array(data =
[[ 0  1]
[ 2 --]],
mask =
[[False False]
[False  True]],
fill_value = 999999)

>>> ma.transpose(x)
masked_array(data =
[[ 0  2]
[ 1 --]],
mask =
[[False False]
[False  True]],
fill_value = 999999)
```

**swapaxes()**
Return a view of the array with axis1 and axis2 interchanged.

Refer to **numpy.swapaxes** for full documentation.

See Also:

**numpy.swapaxes**
equivalent function

**transpose()**
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]])
```

**Changing the number of dimensions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.atleast_1d(*arys)</code></td>
<td>Convert inputs to arrays with at least one dimension. View inputs as arrays with at least one dimension. Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.</td>
</tr>
<tr>
<td><code>ma.atleast_2d(*arys)</code></td>
<td>View inputs as arrays with at least two dimensions.</td>
</tr>
<tr>
<td><code>ma.atleast_3d(*arys)</code></td>
<td>View inputs as arrays with at least three dimensions.</td>
</tr>
<tr>
<td><code>ma.expand_dims(x, axis)</code></td>
<td>Expand the shape of an array.</td>
</tr>
<tr>
<td><code>ma.squeeze(a)</code></td>
<td>Remove single-dimensional entries from the shape of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.squeeze()</code></td>
<td>Remove single-dimensional entries from the shape of <code>a</code>.</td>
</tr>
<tr>
<td><code>ma.column_stack(tup)</code></td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td><code>ma.concatenate(arrays[, axis])</code></td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
<tr>
<td><code>ma.dstack(tup)</code></td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td><code>ma.hstack(tup)</code></td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td><code>ma.hsplit(ary, indices_or_sections)</code></td>
<td>Split an array into multiple sub-arrays horizontally (column-wise).</td>
</tr>
<tr>
<td><code>ma.mr_</code></td>
<td>Translate slice objects to concatenation along the first axis.</td>
</tr>
<tr>
<td><code>ma.row_stack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
<tr>
<td><code>ma.vstack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

`atleast_1d(*arys)`
Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**

- `array1, array2, ...` : array_like
One or more input arrays.

**Returns**

- **ret** : ndarray
  
  An array, or sequence of arrays, each with `a.ndim >= 1`. Copies are made only if necessary.

**See Also:**

- `atleast_2d`, `atleast_3d`

**Examples**

```python
>>> np.atleast_1d(1.0)
array([1.])

>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])

>>> np.atleast_1d(x) is x
True

>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]
```

**atleast_2d(**`*arys*`**)

View inputs as arrays with at least two dimensions.

**Parameters**

- **array1, array2, ...** : array_like
  
  One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns**

- **res, res2, ...** : ndarray
  
  An array, or tuple of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

**See Also:**

- `atleast_1d`, `atleast_3d`

**Examples**

```python
>>> np.atleast_2d(3.0)
array([[3.]])

>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[0., 1., 2.]])

>>> np.atleast_2d(x).base is x
True
```

3.20. Masked array operations
>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]

atleast_3d(*arys)

View inputs as arrays with at least three dimensions.

Parameters
array1, array2, ... : array_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

Returns
res1, res2, ... : ndarray

An array, or tuple of arrays, each with `a.ndim` >= 3. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `N` becomes a view of shape `(1, N, 1)`. A 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

See Also:
atleast_1d, atleast_2d

Examples

>>> np.atleast_3d(3.0)
array([[3.]])

>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)

>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x
True

>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print(arr, arr.shape)
...[[[1]]] (1, 2, 1)
[[[1]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)

expand_dims(x, axis)

Expand the shape of an array.

Expands the shape of the array by including a new axis before the one specified by the `axis` parameter. This function behaves the same as `numpy.expand_dims` but preserves masked elements.

See Also:
**numpy.expand_dims**

Equivalent function in top-level NumPy module.

**Examples**

```python
given code snippet
```

```python
given code snippet
```
The same result can be achieved using slicing syntax with `np.newaxis`.

```python
given code snippet
```

**squeeze**

Remove single-dimensional entries from the shape of an array.

**Parameters**

- `a`: array_like
  
  Input data.

**Returns**

- `squeezed`: ndarray
  
  The input array, but with with all dimensions of length 1 removed. Whenever possible, a view on `a` is returned.

**Examples**

```python
given code snippet
```

```python
given code snippet
```

Refer to `numpy.squeeze` for full documentation.

**See Also:**

3.20. Masked array operations
numpy.squeeze

equivalent function

column_stack()

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are
stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters
tup : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

Returns
stacked : 2-D array

The array formed by stacking the given arrays.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

concatenate(arrays, axis=0)

Concatenate a sequence of arrays along the given axis.

Parameters
arrays : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to axis
(the first, by default).

axis : int, optional

The axis along which the arrays will be joined. Default is 0.

Returns
result : MaskedArray

The concatenated array with any masked entries preserved.

See Also:

numpy.concatenate

Equivalent function in the top-level NumPy module.

Examples
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)

>>> a
masked_array(data = [0 -- 2],
              mask = [False True False],
             fill_value = 999999)

>>> b
masked_array(data = [2 3 4],
              mask = False,
             fill_value = 999999)

>>> ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
              mask = [False True False False False False],
             fill_value = 999999)

**dstack()**

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by *dsplit*. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

**Parameters**

- **tup**: sequence of arrays

  Arrays to stack. All of them must have the same shape along all but the third axis.

**Returns**

- **stacked**: ndarray

  The array formed by stacking the given arrays.

**See Also**:

- **vstack**: Stack along first axis.
- **hstack**: Stack along second axis.
- **concatenate**: Join arrays.
- **dsplit**: Split array along third axis.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
```

3.20. Masked array operations
array([[1, 2],
       [2, 3],
       [3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a, b))
array([[1, 2],
       [2, 3],
       [3, 4]])

hstack()

Stack arrays in sequence horizontally (column wise).
Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

Parameters

tup : sequence of ndarrays
    All arrays must have the same shape along all but the second axis.

Returns

stacked : ndarray
    The array formed by stacking the given arrays.

See Also:

vstack
    Stack arrays in sequence vertically (row wise).

dstack
    Stack arrays in sequence depth wise (along third axis).

concatenate
    Join a sequence of arrays together.

hsplit
    Split array along second axis.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

>>> a = np.array((1, 2, 3))
>>> b = np.array((2, 3, 4))
>>> np.hstack((a, b))
array([1, 2, 3, 2, 3, 4])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a, b))
array([[1, 2],
       [2, 3],
       [3, 4]])
hsplit()

Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the split documentation. hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.

See Also:

split
Split an array into multiple sub-arrays of equal size.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]]
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
 array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]]))
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
 array([[ 3.],
        [ 7.],
        [11.],
        [15.]]),
 array([], dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.

```python
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
>>> np.hsplit(x, 2)
[array([[[ 0.,  1.]],
        [[ 4.,  5.]]),
 array([[[ 2.,  3.]],
        [[ 6.,  7.]]])]
```

3.20. Masked array operations 707
mr_()
Translate slice objects to concatenation along the first axis.
This is the masked array version of lib.index_tricks.RClass.

See Also:
lib.index_tricks.RClass

Examples

```python
>>> np.ma.mr_[np.ma.array([1, 2, 3]), 0, 0, np.ma.array([4, 5, 6])]
array([1, 2, 3, 0, 0, 4, 5, 6])
```

row_stack()

Stack arrays in sequence vertically (row wise).
Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

Parameters
tup : sequence of ndarrays
   Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns
stacked : ndarray
   The array formed by stacking the given arrays.

See Also:

hstack
Stack arrays in sequence horizontally (column wise).
dstack
Stack arrays in sequence depth wise (along third dimension).
concatenate
Join a sequence of arrays together.
vsplit
Split array into a list of multiple sub-arrays vertically.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

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

**vstack()**

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by `vsplit`.

**Parameters**

`tup` : sequence of ndarrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

**Returns**

`stacked` : ndarray

The array formed by stacking the given arrays.

**See Also:**

`hstack` : Stack arrays in sequence horizontally (column wise).

`dstack` : Stack arrays in sequence depth wise (along third dimension).

`concatenate` : Join a sequence of arrays together.

`vsplit` : Split array into a list of multiple sub-arrays vertically.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

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

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a, b))
```
Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.column_stack(tup)</code></td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td><code>ma.concatenate(arrays[, axis])</code></td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
<tr>
<td><code>ma.dstack(tup)</code></td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td><code>ma.hstack(tup)</code></td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td><code>ma.vstack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

column_stack()

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

**Parameters**

- `tup`: sequence of 1-D or 2-D arrays.

  Arrays to stack. All of them must have the same first dimension.

**Returns**

- `stacked`: 2-D array

  The array formed by stacking the given arrays.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> a = np.array([[1, 2, 3]])
>>> b = np.array([[2, 3, 4]])
>>> np.column_stack((a, b))
array([[1, 2, 3],
       [2, 3, 4]])
```
**concatenate** *(arrays, axis=0)*

Concatenate a sequence of arrays along the given axis.

**Parameters**
- **arrays**: sequence of array_like
  The arrays must have the same shape, except in the dimension corresponding to `axis` (the first, by default).
- **axis**: int, optional
  The axis along which the arrays will be joined. Default is 0.

**Returns**
- **result**: MaskedArray
  The concatenated array with any masked entries preserved.

**See Also**:
- `numpy.concatenate`
  Equivalent function in the top-level NumPy module.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
         fill_value = 999999)
>>> b
masked_array(data = [2 3 4],
             mask = False,
         fill_value = 999999)
>>> ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
         fill_value = 999999)
```

**dstack()**

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by `dsplit`. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

**Parameters**
- **tup**: sequence of arrays
  Arrays to stack. All of them must have the same shape along all but the third axis.

**Returns**
- **stacked**: ndarray
  The array formed by stacking the given arrays.
See Also:

vstack
Stack along first axis.

hstack
Stack along second axis.

concatenate
Join arrays.

dslog
Split array along third axis.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

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

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

hstack()

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

Parameters
tup : sequence of ndarrays
    All arrays must have the same shape along all but the second axis.

Returns
stacked : ndarray
    The array formed by stacking the given arrays.

See Also:

vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third axis).
**concatenate**  
Join a sequence of arrays together.

**hsplit**  
Split array along second axis.

**Notes**  
The function is applied to both the _data and the _mask, if any.

**Examples**

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

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

**vstack()**  

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by **vsplit**.

**Parameters**

- **tup**: sequence of ndarrays
  
  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

**Returns**

- **stacked**: ndarray
  
  The array formed by stacking the given arrays.

**See Also:**

**hstack**  
Stack arrays in sequence horizontally (column wise).

**dstack**  
Stack arrays in sequence depth wise (along third dimension).

**concatenate**  
Join a sequence of arrays together.

**vsplit**  
Split array into a list of multiple sub-arrays vertically.

**Notes**  
The function is applied to both the _data and the _mask, if any.
Examples

>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])

3.20.5 Operations on masks

Creating a mask

make_mask (m[, copy, shrink, flag, ...])
Create a boolean mask from an array.

Return m as a boolean mask, creating a copy if necessary or requested. The function can accept any sequence that is convertible to integers, or nomask. Does not require that contents must be 0s and 1s, values of 0 are interpreted as False, everything else as True.

Parameters

- m : array_like
  Potential mask.
- copy : bool, optional
  Whether to return a copy of m (True) or m itself (False).
- shrink : bool, optional
  Whether to shrink m to nomask if all its values are False.
- flag : bool, optional
  Deprecated equivalent of shrink.
- dtype : dtype, optional
  Data-type of the output mask. By default, the output mask has a dtype of MaskType (bool). If the dtype is flexible, each field has a boolean dtype.

Returns

- result : ndarray
A boolean mask derived from \(m\).

**Examples**

```python
>>> import numpy.ma as ma
>>> m = [True, False, True, True]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
>>> m = [1, 0, 1, 1]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
>>> m = [1, 0, 2, -3]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
```

Effect of the `shrink` parameter.

```python
>>> m = np.zeros(4)
>>> m
array([ 0., 0., 0., 0.])
>>> ma.make_mask(m)
False
>>> ma.make_mask(m, shrink=False)
array([False, False, False, False], dtype=bool)
```

Using a flexible `dtype`.

```python
>>> m = [1, 0, 1, 1]
>>> n = [0, 1, 0, 0]
>>> arr = []
>>> for man, mouse in zip(m, n):
...     arr.append((man, mouse))
>>> arr
[(1, 0), (0, 1), (1, 0), (1, 0)]
>>> dtype = np.dtype({'names':['man', 'mouse'],
                   'formats':[np.int, np.int]})
>>> arr = np.array(arr, dtype=dtype)
>>> arr
array([(1, 0), (0, 1), (1, 0), (1, 0)],
      dtype=[('man', '<i4'), ('mouse', '<i4')])
>>> ma.make_mask(arr, dtype=dtype)
array([(True, False), (False, True), (True, False), (True, False)],
      dtype=[('man', '|b1'), ('mouse', '|b1')])
```

**make_mask_none (newshape, dtype=\(None\))**

Return a boolean mask of the given shape, filled with False.

This function returns a boolean ndarray with all entries False, that can be used in common mask manipulations. If a complex dtype is specified, the type of each field is converted to a boolean type.

**Parameters**

- **newshape** : tuple

  A tuple indicating the shape of the mask.

- **dtype** : {\(None, \) dtype, \(optional\)}

  If None, use a MaskType instance. Otherwise, use a new datatype with the same fields as `dtype`, converted to boolean types.
Returns

result : ndarray

An ndarray of appropriate shape and dtype, filled with False.

See Also:

make_mask
Create a boolean mask from an array.

make_mask_descr
Construct a dtype description list from a given dtype.

Examples

>>> import numpy.ma as ma
>>> ma.make_mask_none((3,))
array([False, False, False], dtype=bool)

Defining a more complex dtype.

>>> dtype = np.dtype({'names': ['foo', 'bar'],
                   'formats': [np.float32, np.int]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i4'))
>>> ma.make_mask_none((3,), dtype=dtype)
array([(False, False), (False, False), (False, False)],
      dtype=[('foo', '|b1'), ('bar', '|b1')])

mask_or (m1, m2, copy=False, shrink=True)
Combine two masks with the logical_or operator.

The result may be a view on m1 or m2 if the other is nomask (i.e. False).

Parameters

m1, m2 : array_like
Input masks.

copy : bool, optional
If copy is False and one of the inputs is nomask, return a view of the other input mask.
Defaults to False.

shrink : bool, optional
Whether to shrink the output to nomask if all its values are False. Defaults to True.

Returns

mask : output mask
The result masks values that are masked in either m1 or m2.

Raises

ValueError :
If m1 and m2 have different flexible dtypes.

Examples
```python
>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([ True,  True,  True, False], dtype=bool)
```

**make_mask_descr** *(ndtype)*

Construct a dtype description list from a given dtype.

Returns a new dtype object, with the type of all fields in *ndtype* to a boolean type. Field names are not altered.

**Parameters**

*ndtype* : dtype

The dtype to convert.

**Returns**

*result* : dtype

A dtype that looks like *ndtype*, the type of all fields is boolean.

**Examples**

```python
>>> import numpy.ma as ma
>>> dtype = np.dtype({'names':['foo', 'bar'],
                  'formats':[np.float32, np.int]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i4')])
>>> ma.make_mask_descr(dtype)
dtype([('foo', '|b1'), ('bar', '|b1')])
>>> ma.make_mask_descr(np.float32)
<type 'numpy.bool_'>
```

## Accessing a mask

<table>
<thead>
<tr>
<th><strong>ma.getmask</strong> <em>(a)</em></th>
<th>Return the mask of a masked array, or nomask.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ma.getmaskarray</strong> <em>(arr)</em></td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td><strong>ma.masked_array.mask</strong></td>
<td>Mask</td>
</tr>
</tbody>
</table>

**getmask** *(a)*

Return the mask of a masked array, or nomask.

Return the mask of *a* as an ndarray if *a* is a *MaskedArray* and the mask is not *nomask*, else return *nomask*. To guarantee a full array of booleans of the same shape as *a*, use *getmaskarray*.

**Parameters**

*a* : array_like

Input *MaskedArray* for which the mask is required.

**See Also:**

**getdata**

Return the data of a masked array as an ndarray.

**getmaskarray**

Return the mask of a masked array, or full array of False.
Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1, 2], [3, 4]], 2)
>>> a
masked_array(data =
   [[1  --]
    [3  4]],
  mask =
   [[False   True]
    [False False]],
  fill_value=999999)
>>> ma.getmask(a)
array([[False,  True],
       [False, False]], dtype=bool)

Equivalently use the `MaskedArray mask` attribute.

```python
>>> a.mask
array([[False,  True],
       [False, False]], dtype=bool)

Result when mask == nomask

```python
>>> b = ma.masked_array([[1, 2], [3, 4]])
>>> b
masked_array(data =
   [[1   2]
    [3  4]],
  mask =
   False,
  fill_value=999999)
>>> ma.nomask
False
>>> ma.getmask(b) == ma.nomask
True
>>> b.mask == ma.nomask
True
```

**getmaskarray** *(arr)*

Return the mask of a masked array, or full boolean array of False.

Return the mask of *arr* as an ndarray if *arr* is a `MaskedArray` and the mask is not `nomask`, else return a full boolean array of False of the same shape as *arr*.

Parameters

- **arr**: array_like
  
  Input `MaskedArray` for which the mask is required.

See Also:

- **getmask**
  
  Return the mask of a masked array, or nomask.

- **getdata**
  
  Return the data of a masked array as an ndarray.
Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False  True]
[False False]],
fill_value=999999)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
[3 4]],
mask = False,
fill_value=999999)
```

```python
>>> ma.getmaskarray(a)
array([[False, True],
       [False, False]], dtype=bool)
```

```python
>>> ma.getmaskarray(b)
array([[False, False],
       [False, False]], dtype=bool)
```

Finding masked data

```
ma.flatnotmasked_contiguous(a)  Find contiguous unmasked data in a masked array along the given axis.
ma.flatnotmasked_edges(a)       Find the indices of the first and last unmasked values.
ma.notmasked_contiguous(a[, axis]) Find contiguous unmasked data in a masked array along the given axis.
ma.notmasked_edges(a[, axis])   Find the indices of the first and last unmasked values along an axis.
```

### flatnotmasked_contiguous (a)

Find contiguous unmasked data in a masked array along the given axis.

**Parameters**

- `a` : array
  
The input array.

**Returns**

- `slice_list` : list
  
  A sorted sequence of slices (start index, end index).

**See Also:**

- `flatnotmasked_edges, notmasked_contiguous, notmasked_edges`
**Notes**

Only accepts 2-D arrays at most.

**Examples**

```python
>>> a = np.arange(10)
>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> ma = np.ma.array(a, mask=mask)
>>> np.array(ma[~ma.mask])
array([3, 4, 6, 7, 8])

>>> np.ma.extras.flatnotmasked_contiguous(ma)
[slice(3, 4, None), slice(6, 8, None)]
>>> ma = np.ma.array(a, mask=np.ones_like(a))
>>> np.ma.extras.flatnotmasked_contiguous(ma)
```

**flatnotmasked_edges**(*a*)

Find the indices of the first and last unmasked values.

Expects a 1-D *MaskedArray*, returns None if all values are masked.

**Parameters**

*arr* : array_like

Input 1-D *MaskedArray*

**Returns**

*edges* : ndarray or None

The indices of first and last non-masked value in the array. Returns None if all values are masked.

**See Also:**

*flatnotmasked_contiguous*, *notmasked_contiguous*, *notmasked_edges*

**Notes**

Only accepts 1-D arrays.

**Examples**

```python
>>> a = np.arange(10)
>>> mask = (a < 3) | (a > 8) | (a == 5)

>>> ma = np.ma.array(a, mask=m)
>>> np.array(ma[~ma.mask])
array([3, 4, 6, 7, 8])

>>> flatnotmasked_edges(ma)
array([3, 8])

>>> ma = np.ma.array(a, mask=np.ones_like(a))
>>> print flatnotmasked_edges(ma)
None
```
**notmasked_contiguous** *(a, axis=None)*  
Find contiguous unmasked data in a masked array along the given axis.

**Parameters**
- **a**: array_like  
The input array.
- **axis**: int, optional  
Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

**Returns**
- **endpoints**: list  
A list of slices (start and end indexes) of unmasked indexes in the array.

**See Also:**
- flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges

**Notes**
Only accepts 2-D arrays at most.

**Examples**

```python
g = np.arange(9).reshape((3, 3))  
m = np.zeros_like(g)  
m[1:, 1:] = 1  
ma = np.ma.array(g, mask=m)  
np.array(ma[~ma.mask])  
array([0, 1, 2, 3, 6])

np.ma.extras.notmasked_contiguous(ma)  
[slice(0, 3, None), slice(6, 6, None)]
```

**notmasked_edges** *(a, axis=None)*
Find the indices of the first and last unmasked values along an axis.  
If all values are masked, return None. Otherwise, return a list of two tuples, corresponding to the indices of the first and last unmasked values respectively.

**Parameters**
- **a**: array_like  
The input array.
- **axis**: int, optional  
Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

**Returns**
- **edges**: ndarray or list  
An array of start and end indexes if there are any masked data in the array. If there are no masked data in the array, edges is a list of the first and last index.
See Also:

flatnotmasked_contiguous, flatnotmasked_edges, notmasked_contiguous

Examples

```python
>>> a = np.arange(9).reshape((3, 3))
>>> m = np.zeros_like(a)
>>> m[1:, 1:] = 1

>>> ma = np.ma.array(a, mask=m)
>>> np.array(ma[~ma.mask])
array([0, 1, 2, 3, 6])

>>> np.ma.extras.notmasked_edges(ma)
array([0, 6])
```

Modifying a mask

```python
ma.mask_cols(a, axis=None)
ma.mask_or(m1, m2[, copy, shrink])
ma.mask_rowcols(a[, axis])
ma.harden_mask(self)
ma.soften_mask(self)
ma.MaskedArray.harden_mask(self)
ma.MaskedArray.soften_mask(self)
ma.MaskedArray.shrink_mask(self)
ma.MaskedArray.unshare_mask(self)
```

mask_cols (a, axis=0)

Mask columns of a 2D array that contain masked values.

This function is a shortcut to mask_rowcols with axis equal to 1.

See Also:

mask_rowcols

Mask rows and/or columns of a 2D array.

masked_where

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])

>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
```python
[[0 0 0]
[0 -- 0]
[0 0 0]],
    mask =
[[False False False]
[False True False]
[False False False]],
    fill_value=999999)
>>> ma.mask_cols(a)
masked_array(data =
[[0 -- 0]
[0 -- 0]
[0 -- 0]],
    mask =
[[False True False]
[False True False]
[False True False]],
    fill_value=999999)
```

```python
mask_or (m1, m2, copy=False, shrink=True)
Combine two masks with the logical_or operator.
The result may be a view on m1 or m2 if the other is nomask (i.e. False).

Parameters

m1, m2 : array_like
Input masks.

copy : bool, optional
If copy is False and one of the inputs is nomask, return a view of the other input mask. Defaults to False.

shrink : bool, optional
Whether to shrink the output to nomask if all its values are False. Defaults to True.

Returns

mask : output mask
The result masks values that are masked in either m1 or m2.

Raises

ValueError :
If m1 and m2 have different flexible dtypes.

Examples

```python
>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([ True, True, True, False], dtype=bool)
```

```python
mask_rowcols (a, axis=None)
Mask rows and/or columns of a 2D array that contain masked values.

Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected using the axis parameter.

• If axis is None, rows and columns are masked.
```
• If `axis` is 0, only rows are masked.
• If `axis` is 1 or -1, only columns are masked.

**Parameters**

- `a`: array_like, MaskedArray
  The array to mask. If not a MaskedArray instance (or if no array elements are masked).
  The result is a MaskedArray with `mask` set to `nomask` (False). Must be a 2D array.

- `axis`: int, optional
  Axis along which to perform the operation. If None, applies to a flattened version of the array.

**Returns**

- `a`: MaskedArray
  A modified version of the input array, masked depending on the value of the `axis` parameter.

**Raises**

- `NotImplementedError`
  If input array `a` is not 2D.

**See Also:**

- `mask_rows`
  Mask rows of a 2D array that contain masked values.

- `mask_cols`
  Mask cols of a 2D array that contain masked values.

- `masked_where`
  Mask where a condition is met.

**Notes**

The input array's mask is modified by this function.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
[[0 0 0]
 [0 -- 0]
 [0 0 0]],
mask = [[False False False]
 [False True False]
 [False False False]],
fill_value=999999)
```
>>> ma.mask_rowcols(a)
masked_array(data =
[[0 -- 0]
[-- -- --]
[0 -- 0]],
  mask =
[[False  True False]
 [ True  True  True]
 [False  True False]],
   fill_value=999999)

mask_rows (a, axis=None)
Mask rows of a 2D array that contain masked values.

This function is a shortcut to mask_rowcols with axis equal to 0.

See Also:

mask_rowcols
Mask rows and/or columns of a 2D array.

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
[[0 0 0]
[0 -- 0]
[0 0 0]],
  mask =
[[False False False]
 [False  True False]
 [False False False]],
   fill_value=999999)
>>> ma.mask_rows(a)
masked_array(data =
[[0 0 0]
[-- -- --]
[0 0 0]],
  mask =
[[False False False]
 [ True  True  True]
 [False False False]],
   fill_value=999999)

harden_mask()
Force the mask to hard.
Whether the mask of a masked array is hard or soft is determined by its `hardmask` property. `harden_mask` sets `hardmask` to True.

**See Also:**  
`hardmask`

`soften_mask()`  
Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its `hardmask` property. `soften_mask` sets `hardmask` to False.

**See Also:**  
`hardmask`

`harden_mask()`  
Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its `hardmask` property. `harden_mask` sets `hardmask` to True.

**See Also:**  
`hardmask`

`soften_mask()`  
Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its `hardmask` property. `soften_mask` sets `hardmask` to False.

**See Also:**  
`hardmask`

`shrink_mask()`  
Reduce a mask to nomask when possible.

**Parameters**  
None:

**Returns**  
None:

**Examples**

```python  
>>> x = np.ma.array([[1, 2], [3, 4]], mask=[0]*4)  
>>> x.mask  
array([False, False],  
[False, False], dtype=bool)  
>>> x.shrink_mask()  
>>> x.mask  
False
```

`unshare_mask()`  
Copy the mask and set the sharedmask flag to False.

Whether the mask is shared between masked arrays can be seen from the `sharedmask` property. `unshare_mask` ensures the mask is not shared. A copy of the mask is only made if it was shared.

**See Also:**  
`sharedmask`
3.20.6 Conversion operations

> to a masked array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.asarray(a[, dtype, order])</code></td>
<td>Convert the input to a masked array of the given data-type.</td>
</tr>
<tr>
<td><code>ma.asanyarray(a[, dtype])</code></td>
<td>Convert the input to a masked array, conserving subclasses.</td>
</tr>
<tr>
<td><code>ma.fix_invalid(a[, mask, copy, fill_value])</code></td>
<td>Return input with invalid data masked and replaced by a fill value.</td>
</tr>
<tr>
<td><code>ma.masked_equal(x[, value, copy])</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater(x[, value, copy])</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater_equal(x[, value, copy])</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_inside(x[, v1, v2[, copy]])</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_invalid(a[, copy])</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>ma.masked_less(x[, value, copy])</code></td>
<td>Mask an array where less than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_less_equal(x[, value, copy])</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_not_equal(x[, value, copy])</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_object(x[, value[, copy, shrink]])</code></td>
<td>Mask the array x where the data are exactly equal to value.</td>
</tr>
<tr>
<td><code>ma.masked_outside(x[, v1, v2[, copy]])</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_values(x[, value, rtol, atol, copy,...])</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>ma.masked_where(condition, a[, copy])</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
</tbody>
</table>

**asarray** *(a, dtype=None, order=None)*

Convert the input to a masked array of the given data-type. No copy is performed if the input is already an ndarray. If a is a subclass of MaskedArray, a base class MaskedArray is returned.

**Parameters**

- **a**: array_like
  
  Input data, in any form that can be converted to a masked array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists, ndarrays and masked arrays.

- **dtype**: dtype, optional
  
  By default, the data-type is inferred from the input data.

- **order**: {'C', 'F'}, optional
  
  Whether to use row-major (‘C’) or column-major (‘FORTRAN’) memory representation. Default is ‘C’.

**Returns**

- **out**: MaskedArray
  
  Masked array interpretation of a.

**See Also**:

- **asanyarray**
  
  Similar to asarray, but conserves subclasses.

**Examples**
```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asarray(x)
masked_array(data=
[ [ 0.  1.  2.  3.  4.]
[ 5.  6.  7.  8.  9.] ],
    mask=False,
    fill_value=1e+20)
>>> type(np.ma.asarray(x))
<class 'numpy.ma.core.MaskedArray'>
```

**asanyarray** *(a, dtype=None)*

Convert the input to a masked array, conserving subclasses.

If `a` is a subclass of `MaskedArray`, its class is conserved. No copy is performed if the input is already an `ndarray`.

**Parameters**

- **a** : array_like
  Input data, in any form that can be converted to an array.

- **dtype** : dtype, optional
  By default, the data-type is inferred from the input data.

- **order** : {'C', 'F'}, optional
  Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

**Returns**

- **out** : MaskedArray
  MaskedArray interpretation of `a`.

**See Also:**

- **asarray**
  Similar to `asanyarray`, but does not conserve subclass.

**Examples**

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asanyarray(x)
masked_array(data=
[ [ 0.  1.  2.  3.  4.]
[ 5.  6.  7.  8.  9.] ],
    mask=False,
    fill_value=1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>
```

**fix_invalid** *(a, mask=False, copy=True, fill_value=None)*

Return input with invalid data masked and replaced by a fill value.
Invalid data means values of `nan`, `inf`, etc.

**Parameters**

- `a`: array_like
  
  Input array, a (subclass of) ndarray.

- `copy`: bool, optional
  
  Whether to use a copy of `a` (True) or to fix `a` in place (False). Default is True.

- `fill_value`: scalar, optional
  
  Value used for fixing invalid data. Default is None, in which case the `a.fill_value` is used.

**Returns**

- `b`: MaskedArray

  The input array with invalid entries fixed.

**Notes**

A copy is performed by default.

**Examples**

```python
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [-- -1.0 nan inf],
             mask = [ True False False False],
             fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data = [-- -1.0 -- --],
             mask = [ True False True True],
             fill_value = 1e+20)
```

```python
>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.00000000e+00, -1.00000000e+00,  1.00000000e+20,  1.00000000e+20])
>>> x.data
array([ 1., -1., NaN, Inf])
```

**masked_equal** (x, value, copy=True)

Mask an array where equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x == value)`. For floating point arrays, consider using `masked_values(x, value)`.

**See Also:**

- `masked_where`:
  
  Mask where a condition is met.

- `masked_values`:
  
  Mask using floating point equality.
Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
              mask = [False False True False],
              fill_value=999999)

masked_greater (x, value, copy=True)
Mask an array where greater than a given value.

This function is a shortcut to masked_where, with condition = (x > value).

See Also:

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data = [0 1 2 --],
              mask = [False False False True],
              fill_value=999999)

masked_greater_equal (x, value, copy=True)
Mask an array where greater than or equal to a given value.

This function is a shortcut to masked_where, with condition = (x >= value).

See Also:

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 -- --],
              mask = [False False True True],
              fill_value=999999)

masked_inside (x, v1, v2, copy=True)
Mask an array inside a given interval.

Shortcut to masked_where, where condition is True for x inside the interval [v1,v2] (v1 <= x <= v2). The boundaries v1 and v2 can be given in either order.
See Also:

masked_where

Mask where a condition is met.

Notes

The array $x$ is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
       mask = [False False True True False False],
      fill_value=1e+20)

The order of $v1$ and $v2$ doesn’t matter.

``` python
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
       mask = [False False True True False False],
      fill_value=1e+20)
```

masked_invalid($a$, copy=True)

Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to masked_where, with $condition = \sim(np.isfinite(a))$. Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

See Also:

masked_where

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0., 1., NaN, Inf, 4.])
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
       mask = [False False True True False],
      fill_value=1e+20)
```

masked_less($x$, value, copy=True)

Mask an array where less than a given value.

This function is a shortcut to masked_where, with $condition = (x < value)$.

See Also:
**masked_where**
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
mapped_array(data = [-- -- 2 3],
mask = [ True True False False],
fill_value=999999)
```

**masked_less_equal** *(x, value, copy=True)*
Mask an array where less than or equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x <= value)`.

**See Also:**

**masked_where**
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
mapped_array(data = [-- -- -- 3],
mask = [ True True True False],
fill_value=999999)
```

**masked_not_equal** *(x, value, copy=True)*
Mask an array where not equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x != value)`.

**See Also:**

**masked_where**
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
mapped_array(data = [-- -- 2 --],
mask = [ True True False True],
fill_value=999999)
```

**masked_object** *(x, value, copy=True, shrink=True)*
Mask the array `x` where the data are exactly equal to `value`.

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_object(a, 2)
mapped_array(data = [-- -- -- 3],
mask = [ True True True False],
fill_value=999999)
```
This function is similar to `masked_values`, but only suitable for object arrays: for floating point, use `masked_values` instead.

**Parameters**
- `x`: array_like
  Array to mask
- `value`: object
  Comparison value
- `copy`: {True, False}, optional
  Whether to return a copy of `x`.
- `shrink`: {True, False}, optional
  Whether to collapse a mask full of False to nomask

**Returns**
- `result`: MaskedArray
  The result of masking `x` where equal to `value`.

**See Also:**
- `masked_where`
  Mask where a condition is met.
- `masked_equal`
  Mask where equal to a given value (integers).
- `masked_values`
  Mask using floating point equality.

**Examples**

```python
>>> import numpy.ma as ma
>>> food = np.array(\['green_eggs', 'ham'\], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> print eat
[-- ham]
>>> # plain ol' ham is boring
>>> fresh_food = np.array(\['cheese', 'ham', 'pineapple'\], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> print eat
[cheese ham pineapple]
```

Note that mask is set to nomask if possible.

```python
>>> eat
masked_array(data = [cheese ham pineapple],
             mask = False,
             fill_value=?)
```

**masked_outside** `x, v1, v2, copy=True`
Mask an array outside a given interval.

Shortcut to `masked_where`, where `condition` is True for `x` outside the interval \([v1,v2] (x < v1)(x > v2)\). The boundaries \(v1\) and \(v2\) can be given in either order.
See Also:

masked_where

Mask where a condition is met.

Notes

The array $x$ is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
             mask = [ True True False False True True],
             fill_value=1e+20)
```

The order of $v1$ and $v2$ doesn’t matter.

```python
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
             mask = [ True True False False True True],
             fill_value=1e+20)
```

masked_values

Mask using floating point equality.

Return a MaskedArray, masked where the data in array $x$ are approximately equal to $value$, i.e. where the following condition is True

$\text{abs}(x - value) \leq atol + rtol \times \text{abs}(value)$

The fill_value is set to $value$ and the mask is set to nomask if possible. For integers, consider using masked_equal.

Parameters

- $x$: array_like
  Array to mask.
- $value$: float
  Masking value.
- $rtol$: float, optional
  Tolerance parameter.
- $atol$: float, optional
  Tolerance parameter (1e-8).
- $copy$: bool, optional
  Whether to return a copy of $x$.
- $shrink$: bool, optional
  Whether to collapse a mask full of False to nomask.

Returns

- $result$: MaskedArray
The result of masking $x$ where approximately equal to $value$.

See Also:

masked_where
   Mask where a condition is met.

masked_equal
   Mask where equal to a given value (integers).

Examples

```python
>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
            mask = [False True False True False],
            fill_value=1.1)
```

Note that `mask` is set to `nomask` if possible.

```python
>>> ma.masked_values(x, 1.5)
masked_array(data = [ 1. 1.1 2. 1.1 3. ],
            mask = False,
            fill_value=1.5)
```

For integers, the fill value will be different in general to the result of `masked_equal`.

```python
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data = [0 1 -- 3 4],
            mask = [False False True False False],
            fill_value=2)
```

```python
>>> ma.masked_equal(x, 2)
masked_array(data = [0 1 -- 3 4],
            mask = [False False True False False],
            fill_value=999999)
```

masked_where `condition, a, copy=True`
   Mask an array where a condition is met.

Return $a$ as an array masked where `condition` is True. Any masked values of $a$ or `condition` are also masked in the output.

Parameters

- **condition**: array_like
  
  Masking condition. When `condition` tests floating point values for equality, consider using `masked_values` instead.

- **a**: array_like
  
  Array to mask.

- **copy**: bool
  
  If True (default) make a copy of $a$ in the result. If False modify $a$ in place and return a view.
Returns

result : MaskedArray

The result of masking \( a \) where \( \text{condition} \) is True.

See Also:

masked_values

Mask using floating point equality.

masked_equal

Mask where equal to a given value.

masked_not_equal

Mask where \( \text{not} \) equal to a given value.

masked_less_equal

Mask where less than or equal to a given value.

masked_greater_equal

Mask where greater than or equal to a given value.

masked_less

Mask where less than a given value.

masked_greater

Mask where greater than a given value.

masked_inside

Mask inside a given interval.

masked_outside

Mask outside a given interval.

masked_invalid

Mask invalid values (NaNs or infs).

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
             mask = [ True True True False],
             fill_value=999999)
```

Mask array \( b \) conditional on \( a \).

```python
>>> b = ['a', 'b', 'c', 'd']
>>> ma.masked_where(a == 2, b)
masked_array(data = ['a' 'b' -- 'd'],
             mask = [False False True False],
             fill_value=N/A)
```

Effect of the \( \text{copy} \) argument.
>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data = [-- -- -- 3],
              mask = [ True True True False],
              fill_value=999999)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
              mask = [False True True False],
              fill_value=999999)

>>> a = np.arange(4)
>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
              mask = [False True True False],
              fill_value=999999)
>>> a
array([99, 1, 2, 3])

When condition or a contain masked values.

>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data = [0 1 -- 3],
              mask = [False False True False],
              fill_value=999999)

>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
>>> b
masked_array(data = [-- 1 2 3],
              mask = [ True False False False],
              fill_value=999999)

>>> ma.masked_where(a == 3, b)
masked_array(data = [-- 1 -- --],
              mask = [ True False True True],
              fill_value=999999)

> to a ndarray

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.compress_cols(a)</code></td>
<td>Suppress whole columns of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td><code>ma.compress_rowcols(x[, axis])</code></td>
<td>Suppress rows and/or columns of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td><code>ma.compress_rows(a)</code></td>
<td>Suppress whole rows of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td><code>ma.compressed(x)</code></td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td><code>ma.filled(a[, fill_value])</code></td>
<td>Return input as an array with masked data replaced by a fill value.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.compressed(self)</code></td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.filled(self[, fill_value])</code></td>
<td>Return a copy of self, where masked values are filled with a fill value.</td>
</tr>
</tbody>
</table>

`compress_cols(a)`

Suppress whole columns of a 2-D array that contain masked values.
This is equivalent to `np.ma.extras.compress_rowcols(a, 1)`, see `extras.compress_rowcols` for details.

**See Also:**

`extras.compress_rowcols`

**compress_rowcols** *(x, axis=None)*

Suppress the rows and/or columns of a 2-D array that contain masked values.

The suppression behavior is selected with the `axis` parameter.

- If `axis` is `None`, both rows and columns are suppressed.
- If `axis` is 0, only rows are suppressed.
- If `axis` is 1 or `-1`, only columns are suppressed.

**Parameters**

- **axis** : int, optional
  
  Axis along which to perform the operation. Default is `None`.

**Returns**

- **compressed_array** : ndarray
  
  The compressed array.

**Examples**

```python
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
... [1, 0, 0],
... [0, 0, 0]])
```

```python
>>> x
masked_array(data =
[[-- 1 2]
[-- 4 5]
[6 7 8]],
mask =
[[ True False False]
[ True False False]
[False False False]],
fill_value = 999999)
```

```python
>>> np.ma.extras.compress_rowcols(x)
array([[7, 8]])
```

```python
>>> np.ma.extras.compress_rowcols(x, 0)
array([[6, 7, 8]])
```

```python
>>> np.ma.extras.compress_rowcols(x, 1)
array([[1, 2],
[4, 5],
[7, 8]])
```

**compress_rows** *(a)*

Suppress whole rows of a 2-D array that contain masked values.

This is equivalent to `np.ma.extras.compress_rowcols(a, 0)`, see `extras.compress_rowcols` for details.

**See Also:**

`extras.compress_rowcols`
**compressed** *(x)*

Return all the non-masked data as a 1-D array.

This function is equivalent to calling the "compressed" method of a `MaskedArray`, see `MaskedArray.compressed` for details.

**See Also:**

`MaskedArray.compressed`

Equivalent method.

**filled** *(a, fill_value=None)*

Return input as an array with masked data replaced by a fill value.

If `a` is not a `MaskedArray`, `a` itself is returned. If `a` is a `MaskedArray` and `fill_value` is None, `fill_value` is set to `a.fill_value`.

**Parameters**

- `a`: MaskedArray or array_like
  An input object.

- `fill_value`: scalar, optional
  Filling value. Default is None.

**Returns**

- `a`: ndarray
  The filled array.

**Examples**

```python
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
... [0, 0, 0]])
>>> x.filled()
array([[999999, 1, 2],
      [999999, 4, 5],
      [ 6, 7, 8]])
```

**compressed()**

Return all the non-masked data as a 1-D array.

**Returns**

- `data`: ndarray
  A new `ndarray` holding the non-masked data is returned.

**Notes**

The result is **not** a `MaskedArray`!

**Examples**

```python
>>> x = np.ma.array(np.arange(5), mask=[0, 1, 0, 1, 0])
>>> x.compressed()
array([0, 1])
>>> type(xcompressed())
<type 'numpy.ndarray'>
```
filled (fill_value=None)

Return a copy of self, where masked values are filled with a fill value.

If fill_value is None, MaskedArray.fill_value is used instead.

Parameters

fill_value : scalar, optional

The value to use for invalid entries. Default is None.

Notes

The result is not a MaskedArray!

Examples

```python
>>> x = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 0, 1], fill_value=-999)
>>> x.filled()
array([1, 2, -999, 4, -999])
>>> type(x.filled())
<type 'numpy.ndarray'>
```

Subclassing is preserved. This means that if the data part of the masked array is a matrix, filled returns a matrix:

```python
>>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.filled()
matrix([[ 1, 999999],
         [999999, 4]])
```

> to another object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.MaskedArray.tofile</td>
<td>Save a masked array to a file in binary format.</td>
</tr>
<tr>
<td>ma.MaskedArray.tolist</td>
<td>Return the data portion of the masked array as a hierarchical Python list.</td>
</tr>
<tr>
<td>ma.MaskedArray.torecords</td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td>ma.MaskedArray.tostring</td>
<td>Return the array data as a string containing the raw bytes in the array.</td>
</tr>
</tbody>
</table>

**tolist** (fill_value=None)

Return the data portion of the masked array as a hierarchical Python list.

Data items are converted to the nearest compatible Python type. Masked values are converted to fill_value. If fill_value is None, the corresponding entries in the output list will be None.

Parameters

fill_value : scalar, optional

The value to use for invalid entries. Default is None.
Returns
result : list

The Python list representation of the masked array.

Examples

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)
>>> x.tolist()
[[1, None, 3], [None, 5, None], [7, None, 9]]
```

```python
>>> x.tolist(-999)
[[1, -999, 3], [-999, 5, -999], [7, -999, 9]]
```

torecords()

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

•the _data field stores the _data part of the array.

•the _mask field stores the _mask part of the array.

Parameters
None:

Returns
record : ndarray

A new flexible-type ndarray with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

Notes
A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value, ...) will be lost.

Examples

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)
```

```python
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
```

```python
>>> print x.toflex()
[[[1, False) (2, True) (3, False)]
 [[4, True) (5, False) (6, True)]
 [(7, False) (8, True) (9, False)])
```

tostring(fill_value=None, order='C')

Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

Parameters
fill_value : scalar, optional

Value used to fill in the masked values. Default is None, in which case MaskedArray.fill_value is used.

order : {'C', 'F', 'A'}, optional
Order of the data item in the copy. Default is ‘C’.

- ‘C’ – C order (row major).
- ‘F’ – Fortran order (column major).
- ‘A’ – Any, current order of array.
- None – Same as ‘A’.

See Also:

ndarray.tostring,tolist, tofile

Notes

As for ndarray.tostring, information about the shape, dtype, etc., but also about fill_value, will be lost.

Examples

```python
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tostring()
'\x01\x00\x00\x00\x0f\x00\x0f\x00\x0f\x0f\x00\x04\x00\x00\x00'
```

Pickling and unpickling

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.dump(a, F)</td>
<td>Pickle a masked array to a file.</td>
</tr>
<tr>
<td>ma.dumps(a)</td>
<td>Return a string corresponding to the pickling of a masked array.</td>
</tr>
<tr>
<td>ma.load(F)</td>
<td>Wrapper around cPickle.load which accepts either a file-like object or a filename.</td>
</tr>
<tr>
<td>ma.loads(strg)</td>
<td>Load a pickle from the current string.</td>
</tr>
</tbody>
</table>


dump(a, F)

Pickle a masked array to a file.

This is a wrapper around cPickle.dump.

**Parameters**

- a : MaskedArray
  
  The array to be pickled.

- F : str or file-like object
  
  The file to pickle a to. If a string, the full path to the file.

dumps(a)

Return a string corresponding to the pickling of a masked array.

This is a wrapper around cPickle.dumps.

**Parameters**

- a : MaskedArray
  
  The array for which the string representation of the pickle is returned.

load(F)

Wrapper around cPickle.load which accepts either a file-like object or a filename.

**Parameters**

- F : str or file
  
  The file or file name to load.

See Also:
dump

Pickle an array

Notes

This is different from \texttt{numpy.load}, which does not use cPickle but loads the NumPy binary .npy format.

loads \((strg)\)

Load a pickle from the current string.

The result of cPickle.loads\((strg)\) is returned.

Parameters

\(strg : \text{str}\)

The string to load.

See Also:

dumps

Return a string corresponding to the pickling of a masked array.

Filling a masked array

\begin{center}
\begin{tabular}{|l|l|}
\hline
\texttt{ma.common\_fill\_value} (a, b) & Return the common filling value of two masked arrays, if any. \\
\texttt{ma.default\_fill\_value} (obj) & Return the default fill value for the argument object. \\
\texttt{ma.maximum\_fill\_value} (obj) & Return the minimum value that can be represented by the dtype of an object. \\
\hline
\texttt{ma.maximum\_fill\_value} (obj) & Return the minimum value that can be represented by the dtype of an object. \\
\texttt{ma.set\_fill\_value} (a, fill\_value) & Set the filling value of a, if a is a masked array. \\
\texttt{ma.MaskedArray.get\_fill\_value} (self) & Return the filling value of the masked array. \\
\texttt{ma.MaskedArray.set\_fill\_value} (self) & \texttt{value}: filling value of the masked array. \\
\texttt{ma.MaskedArray.fill\_value} & Filling value. \\
\end{tabular}
\end{center}

\texttt{common\_fill\_value} \((a, b)\)

Return the common filling value of two masked arrays, if any.

If \(a.fill\_value == b.fill\_value\), return the fill value, otherwise return None.

Parameters

\(a, b : \text{MaskedArray}\)

The masked arrays for which to compare fill values.

Returns

\texttt{fill\_value : scalar or None}

The common fill value, or None.

Examples

\begin{verbatim}
>>> x = np.ma.array([0, 1.], fill_value=3)
>>> y = np.ma.array([0, 1.], fill_value=3)
>>> np.ma.common_fill_value(x, y)
3.0
\end{verbatim}

\texttt{default\_fill\_value} \((obj)\)

Return the default fill value for the argument object.
The default filling value depends on the datatype of the input array or the type of the input scalar:

<table>
<thead>
<tr>
<th>datatype</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>int</td>
<td>999999</td>
</tr>
<tr>
<td>float</td>
<td>1.e20</td>
</tr>
<tr>
<td>complex</td>
<td>1.e20+0j</td>
</tr>
<tr>
<td>object</td>
<td>‘?’</td>
</tr>
<tr>
<td>string</td>
<td>‘N/A’</td>
</tr>
</tbody>
</table>

**Parameters**

**obj**: ndarray, dtype or scalar

The array data-type or scalar for which the default fill value is returned.

**Returns**

**fill_value**: scalar

The default fill value.

**Examples**

```python
>>> np.ma.default_fill_value(1)
999999
>>> np.ma.default_fill_value(np.array([1.1, 2., np.pi]))
1e+20
>>> np.ma.default_fill_value(np.dtype(complex))
(1e+20+0j)
```

**maximum_fill_value**(obj)

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

**Parameters**

**obj**: {ndarray, dtype}

An object that can be queried for it’s numeric type.

**Returns**

**val**: scalar

The minimum representable value.

**Raises**

**TypeError**:

If *obj* isn’t a suitable numeric type.

**See Also:**

**minimum_fill_value**

The inverse function.

**set_fill_value**

Set the filling value of a masked array.

**MaskedArray.fill_value**

Return current fill value.
Examples

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
An array of numeric data can also be passed.

>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf
```

maximum_fill_value(obj)

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

Parameters

- obj : {ndarray, dtype}
  An object that can be queried for it's numeric type.

Returns

- val : scalar
  The minimum representable value.

 Raises

- TypeError
  If obj isn’t a suitable numeric type.

See Also:

minimum_fill_value

The inverse function.

set_fill_value

Set the filling value of a masked array.

MaskedArray.fill_value

Return current fill value.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
An array of numeric data can also be passed.
```
set_fill_value(a, fill_value)
Set the filling value of a, if a is a masked array.

This function changes the fill value of the masked array a in place. If a is not a masked array, the function returns silently, without doing anything.

Parameters
  a : array_like
      Input array.
  fill_value : dtype
      Filling value. A consistency test is performed to make sure the value is compatible with the dtype of a.

Returns
  None :
      Nothing returned by this function.

See Also:

maximum_fill_value
Return the default fill value for a dtype.

MaskedArray.fill_value
Return current fill value.

MaskedArray.set_fill_value
Equivalent method.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a = ma.masked_where(a < 3, a)
>>> a
masked_array(data = [-- -- -- 3 4],
             mask = [ True True True False False],
             fill_value=999999)
>>> ma.set_fill_value(a, -999)
>>> a
masked_array(data = [-- -- -- 3 4],
             mask = [ True True True False False],
             fill_value=-999)

Nothing happens if a is not a masked array.
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> ma.set_fill_value(a, 100)
>>> a
[0, 1, 2, 3, 4]
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> ma.set_fill_value(a, 100)
>>> a
array([0, 1, 2, 3, 4])

get_fill_value()
Return the filling value of the masked array.

Returns
fill_value : scalar
The filling value.

Examples

>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...    np.ma.array([0, 1], dtype=dt).get_fill_value()
...
999999
999999
1e+20
(1e+20+0j)

>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.get_fill_value()
-inf

set_fill_value(value=None)
Set the filling value of the masked array.

Parameters
value : scalar, optional
The new filling value. Default is None, in which case a default based on the data type is used.

See Also:

ma.set_fill_value
Equivalent function.

Examples

>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.set_fill_value(np.pi)
>>> x.fill_value
3.1415926535897931
Reset to default:

```python
>>> x.set_fill_value()
>>> x.fill_value
1e+20
```

**fill_value**

Filling value.

### 3.20.7 Masked arrays arithmetics

**Arithmetics**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.anom</code></td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td><code>ma.anomalies</code></td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td><code>ma.average</code></td>
<td>Return the weighted average of array over the given axis.</td>
</tr>
<tr>
<td><code>ma.conjugate</code></td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td><code>ma.corrcoef</code></td>
<td>Return correlation coefficients of the input array.</td>
</tr>
<tr>
<td><code>ma.cov</code></td>
<td>Estimate the covariance matrix.</td>
</tr>
<tr>
<td><code>ma.cumsum</code></td>
<td>Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.</td>
</tr>
<tr>
<td><code>ma.cumprod</code></td>
<td>Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.</td>
</tr>
<tr>
<td><code>ma.mean</code></td>
<td>Return the average of the array elements.</td>
</tr>
<tr>
<td><code>ma.median</code></td>
<td>Compute the median along the specified axis.</td>
</tr>
<tr>
<td><code>ma.power</code></td>
<td>Returns element-wise base array raised to power from second array.</td>
</tr>
<tr>
<td><code>ma.prod</code></td>
<td>Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.</td>
</tr>
<tr>
<td><code>ma.std</code></td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td><code>ma.sum</code></td>
<td>Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.</td>
</tr>
<tr>
<td><code>ma.var</code></td>
<td>Compute the variance along the specified axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.anom</code></td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.cumprod</code></td>
<td>Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.cumsum</code></td>
<td>Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.mean</code></td>
<td>Returns the average of the array elements.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.prod</code></td>
<td>Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.std</code></td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.sum</code></td>
<td>Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.var</code></td>
<td>Compute the variance along the specified axis.</td>
</tr>
</tbody>
</table>
anom()  
Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters
axis : int, optional
    Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional
    Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

mean
Compute the mean of the array.

Examples

>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
             mask = False,
             fill_value = 1e+20)

anomalies()
Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters
axis : int, optional
    Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional
    Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

mean
Compute the mean of the array.

Examples

>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
             mask = False,
             fill_value = 1e+20)
average \((a, axis=None, weights=None, returned=False)\)

Return the weighted average of array over the given axis.

**Parameters**

- **a**: array_like
  
  Data to be averaged. Masked entries are not taken into account in the computation.

- **axis**: int, optional
  
  Axis along which the variance is computed. The default is to compute the variance of the flattened array.

- **weights**: array_like, optional
  
  The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of \(a\) along the given axis) or of the same shape as \(a\). If \(weights=None\), then all data in \(a\) are assumed to have a weight equal to one.

- **returned**: bool, optional
  
  Flag indicating whether a tuple \((result, sum of weights)\) should be returned as output (True), or just the result (False). Default is False.

**Returns**

- **average, [sum_of_weights]**: (tuple of) scalar or MaskedArray
  
  The average along the specified axis. When returned is True, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is np.float64 if \(a\) is of integer type, otherwise it is of the same type as \(a\). If returned, sum_of_weights is of the same type as average.

**Examples**

```python
>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25

>>> x = np.ma.arange(6.).reshape(3, 2)
>>> print x
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]]
>>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3],
... returned=True)
>>> print avg
[2.66666666667 3.66666666667]
```

**conjugate**

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

**Parameters**

- **x**: array_like
  
  Input value.

**Returns**

- **y**: ndarray
The complex conjugate of $x$, with same dtype as $y$.

**Examples**

```python
>>> np.conjugate(1+2j)
(1-2j)
```

```python
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j, 0.-0.j],
       [ 0.-0.j, 1.-1.j]])
```

`corrcoef` ($x$, $y$=None, rowvar=True, bias=False, allow_masked=True)

Return correlation coefficients of the input array.

Except for the handling of missing data this function does the same as `numpy.corrcoef`. For more details and examples, see `numpy.corrcoef`.

**Parameters**

- $x$ : array_like
  A 1-D or 2-D array containing multiple variables and observations. Each row of $x$ represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.

- $y$ : array_like, optional
  An additional set of variables and observations. $y$ has the same shape as $x$.

- `rowvar` : bool, optional
  If `rowvar` is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

- `bias` : bool, optional
  Default normalization (False) is by $(N-1)$, where $N$ is the number of observations given (unbiased estimate). If `bias` is 1, then normalization is by $N$.

- `allow_masked` : bool, optional
  If True, masked values are propagated pair-wise: if a value is masked in $x$, the corresponding value is masked in $y$. If False, raises an exception.

**See Also:**

- `numpy.corrcoef` Equivalent function in top-level NumPy module.
- `cov` Estimate the covariance matrix.

`cov` ($x$, $y$=None, rowvar=True, bias=False, allow_masked=True)

Estimate the covariance matrix.

Except for the handling of missing data this function does the same as `numpy.cov`. For more details and examples, see `numpy.cov`.

By default, masked values are recognized as such. If $x$ and $y$ have the same shape, a common mask is allocated: if $x[i,j]$ is masked, then $y[i,j]$ will also be masked. Setting `allow_masked` to False will raise an exception if values are missing in either of the input arrays.
Parameters

x : array_like
A 1-D or 2-D array containing multiple variables and observations. Each row of x
represents a variable, and each column a single observation of all those variables. Also
see rowvar below.

y : array_like, optional
An additional set of variables and observations. y has the same form as x.

rowvar : bool, optional
If rowvar is True (default), then each row represents a variable, with observations in the
columns. Otherwise, the relationship is transposed: each column represents a variable,
while the rows contain observations.

bias : bool, optional
Default normalization (False) is by \((N-1)\), where \(N\) is the number of observations given
(unbiased estimate). If bias is True, then normalization is by \(N\).

allow_masked : bool, optional
If True, masked values are propagated pair-wise: if a value is masked in x, the cor-
responding value is masked in y. If False, raises a ValueError exception when some
values are missing.

Raises

ValueError:
Raised if some values are missing and allow_masked is False.

See Also:
numpy.cov
cumsum()

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the
flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result
will be masked at the same locations.

Parameters

axis : {None, -1, int}, optional
Axis along which the sum is computed. The default (axis = None) is to compute over
the flattened array. axis may be negative, in which case it counts from the last to the
first axis.

dtype : {None, dtype}, optional
Type of the returned array and of the accumulator in which the elements are summed. If
dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a
precision less than that of the default platform integer. In that case, the default platform
integer is used.

out : ndarray, optional
Alternative output array in which to place the result. It must have the same shape and
buffer length as the expected output but the type will be cast if necessary.

Returns
cumsum : ndarray.
A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

**Notes**

The mask is lost if `out` is not a valid `MaskedArray`!

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print(marr.cumsum())
[0 1 3 -- -- -- 9 16 24 33]
```

**cumprod**

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

- `axis`: {None, -1, int}, optional
  Axis along which the product is computed. The default (`axis = None`) is to compute over the flattened array.

- `dtype`: {None, dtype}, optional
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the `dtype` is the same as that of `a`.

- `out`: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

- `cumprod`: ndarray
  A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

**Notes**

The mask is lost if `out` is not a valid `MaskedArray`!

Arithmetic is modular when using integer types, and no error is raised on overflow.

**mean**

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to `numpy.mean` for the full documentation.

**Parameters**

- `a`: array_like
  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.
axis : int, optional
   Axis along which the means are computed. The default is to compute the mean of the
   flattened array.

dtype : dtype, optional
   Type to use in computing the mean. For integer inputs, the default is float64; for floating
   point, inputs it is the same as the input dtype.

out : ndarray, optional
   Alternative output array in which to place the result. It must have the same shape as the
   expected output but the type will be cast if necessary.

Returns
   mean : ndarray, see dtype parameter above
   If out=None, returns a new array containing the mean values, otherwise a reference to
   the output array is returned.

See Also:
   numpy.ma.mean
      Equivalent function.
   numpy.mean
      Equivalent function on non-masked arrays.
   numpy.ma.average
      Weighted average.

Examples

>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
            mask = [False False True],
            fill_value = 999999)
>>> a Mean
1.5

median(a, axis=None, out=None, overwrite_input=False)
   Compute the median along the specified axis.
   Returns the median of the array elements.

Parameters
   a : array_like
      Input array or object that can be converted to an array.
   axis : int, optional
      Axis along which the medians are computed. The default (None) is to compute the
      median along a flattened version of the array.
   out : ndarray, optional
      Alternative output array in which to place the result. It must have the same shape and
      buffer length as the expected output but the type will be cast if necessary.
   overwrite_input : bool, optional
If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is True, and the input is not already an ndarray, an error will be raised.

Returns

median : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return data-type is float64 for integers and floats smaller than float64, or the input data-type, otherwise.

See Also:

mean

Notes

Given a vector V with N non masked values, the median of V is the middle value of a sorted copy of V (Vs) - i.e. Vs[(N-1)/2], when N is odd, or {Vs[N/2 - 1] + Vs[N/2]}/2 when N is even.

Examples

>>> x = np.ma.array(np.arange(8), mask=[0]*4 + [1]*4)

>>> np.ma.extras.median(x)
1.5

>>> x = np.ma.array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)

>>> np.ma.extras.median(x)
2.5

>>> np.ma.extras.median(x, axis=-1, overwrite_input=True)
masked_array(data = [ 2. 5.],
             mask = False,
             fill_value = 1e+20)

power (a, b, third=None)

Returns element-wise base array raised to power from second array.

This is the masked array version of numpy.power. For details see numpy.power.

See Also:

numpy.power

Notes

The out argument to numpy.power is not supported, third has to be None.

prod ()

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

axis : {None, int}, optional

    Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the `dtype` is the same as that of `a`.

**out**: `{None, array}`, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

`product_along_axis`: `{array, scalar}`, see `dtype` parameter above.

Returns an array whose shape is the same as `a` with the specified axis removed. Returns a 0d array when `a` is 1d or `axis=None`. Returns a reference to the specified output array if specified.

**See Also:**

`prod` equivalent function

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> np.prod([1., 2.])
2.0
>>> np.prod([1., 2.], dtype=np.int32)
2
>>> np.prod([[1., 2.], [3., 4.]])
24.0
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([ 2., 12.])
```

**std()**

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

`a` : array_like

Calculate the standard deviation of these values.

`axis` : int, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

`dtype` : dtype, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

`out` : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.
ddof : int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements. By default \( ddof \) is zero.

Returns

standard_deviation : ndarray, see dtype parameter above.

If \( out \) is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

var, mean

numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - \text{x.mean()})*2)} \).

The mean is normally calculated as \( x \text{.sum()} / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of the infinite population. \( ddof=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( ddof=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( std \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( std \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( dtype \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1., 1.])
>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

In single precision, \( std() \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

sum()

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
Parameters

- **axis**: {None, -1, int}, optional
  
  Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out**: {None, ndarray}, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

- **sum_along_axis**: MaskedArray or scalar
  
  An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[1 -- 3]
[-- 5 --]
[7 -- 9]
>>> print x.sum()
25
>>> print x.sum(axis=1)
[4 5 16]
>>> print x.sum(axis=0)
[8 5 12]
>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type 'numpy.int64'>
```

*var()*

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- **a**: array_like
  
  Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

- **axis**: int, optional
  
  Axis along which the variance is computed. The default is to compute the variance of the flattened array.

- **dtype**: dtype, optional
  
  Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.
out : ndarray, optional
   Alternative output array in which to place the result. It must have the same shape as the
   expected output but the type is cast if necessary.

ddof : int, optional
   “Delta Degrees of Freedom”: the divisor used in calculation is $N - \text{ddof}$, where $N$
   represents the number of elements. By default $\text{ddof}$ is zero.

Returns

   variance : ndarray, see dtype parameter above
      If out=None, returns a new array containing the variance; otherwise a reference to the
      output array is returned.

See Also:

   std
   Standard deviation

   mean
   Average

   numpy.doc.ufuncs
      Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., $\text{var} = \text{mean}(\text{abs}(x - x.\text{mean}())^2)$.

The mean is normally calculated as $x.\text{sum}() / N$, where $N = \text{len}(x)$. If, however, $\text{ddof}$ is specified, the
   divisor $N - \text{ddof}$ is used instead. In standard statistical practice, $\text{ddof}=1$ provides an unbiased estimator of
   the variance of the infinite population. $\text{ddof}=0$ provides a maximum likelihood estimate of the variance for
   normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and
   nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the
   input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a
   higher-accuracy accumulator using the $\text{dtype}$ keyword can alleviate this issue.

Examples

   >>> a = np.array([[1,2],[3,4]])
   >>> np.var(a)
   1.25
   >>> np.var(a,0)
   array([ 1., 1.])
   >>> np.var(a,1)
   array([[ 0.25, 0.25]])

In single precision, var() can be inaccurate:

   >>> a = np.zeros((2,512*512), dtype=np.float32)
   >>> a[0,:] = 1.0
   >>> a[1,:] = 0.1
   >>> np.var(a)
   0.20405951142311096
Computing the standard deviation in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

**anom** *(axis=None, dtype=None)*

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**

- **axis** : int, optional
  Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

- **dtype** : dtype, optional
  Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

**See Also:**

**mean**
Compute the mean of the array.

**Examples**

```python
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
             mask = False,
            fill_value = 1e+20)
```

**cumprod** *(axis=None, dtype=None, out=None)*

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

- **axis** : {None, -1, int}, optional
  Axis along which the product is computed. The default (axis = None) is to compute over the flattened array.

- **dtype** : {None, dtype}, optional
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out** : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.
Returns

cumprod : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

cumsum (axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

axis : {None, -1, int}, optional

Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array. axis may be negative, in which case it counts from the last to the first axis.

dtype : {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

cumsum : ndarray.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[ 0  1  3 -- -- --  9 16 24 33]

mean (axis=None, dtype=None, out=None)

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to numpy.mean for the full documentation.
**Parameters**

- `a` : array_like
  
  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.

- `axis` : int, optional
  
  Axis along which the means are computed. The default is to compute the mean of the flattened array.

- `dtype` : dtype, optional
  
  Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

- `out` : ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- `mean` : ndarray, see dtype parameter above
  
  If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See Also:**

- `numpy.ma.mean`
  
  Equivalent function.

- `numpy.mean`
  
  Equivalent function on non-masked arrays.

- `numpy.ma.average`
  
  Weighted average.

**Examples**

```python
>>> a = np.ma.array([1, 2, 3], mask=[False, False, True])
```

```python
>>> a
masked_array(data = [1 2 --],
       mask = [False False True],
      fill_value = 999999)
```

```python
>>> a.mean()
1.5
```

**prod** *(axis=None, dtype=None, out=None)*

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

**Parameters**

- `axis` : {None, int}, optional
  
  Axis over which the product is taken. If None is used, then the product is over all the array elements.

- `dtype` : {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision...
less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

See Also:

prod

equivalent function

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> np.prod([1.,2.])
2.0
>>> np.prod([1.,2.], dtype=np.int32)
2
>>> np.prod([[1.,2.],[3.,4.]])
24.0
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])
```

std (axis=None, dtype=None, out=None, ddof=0)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

a : array_like

Calculate the standard deviation of these values.

axis : int, optional

Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

dtype : dtype, optional

Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

ddof : int, optional

3.20. Masked array operations
Means Delta Degrees of Freedom. The divisor used in calculations is \( N - \text{ddof} \), where \( N \) represents the number of elements. By default \( \text{ddof} \) is zero.

**Returns**

- `standard_deviation` : ndarray, see dtype parameter above.

If `out` is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

**See Also:**

- `var`, `mean`
- `numpy.doc.ufuncs`  
  - Section “Output arguments”

**Notes**

The standard deviation is the square root of the average of the squared deviations from the mean, i.e.,

\[
\text{std} = \sqrt{\text{mean}(\text{abs}(x - x.\text{mean()}^2))}.
\]

The mean is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of the infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( \text{ddof}=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( \text{std} \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( \text{std} \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

**Examples**

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

>>> np.std(a)
1.118033988749895

>>> np.std(a, axis=0)
array([ 1., 1.])

>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

In single precision, \( \text{std}() \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:]=1.0
>>> a[1,:]=0.1

>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

**sum** (axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
Parameters

axis : {None, -1, int}, optional
Axis along which the sum is computed. The default (axis = None) is to compute over
the flattened array.

dtype : {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements
are summed. If dtype has the value None and the type of a is an integer type of precision
less than the default platform integer, then the default platform integer precision is used.
Otherwise, the dtype is the same as that of a.

out : {None, ndarray}, optional
Alternative output array in which to place the result. It must have the same shape and
buffer length as the expected output but the type will be cast if necessary.

Returns

sum_along_axis : MaskedArray or scalar
An array with the same shape as self, with the specified axis removed. If self is a 0-d
array, or if axis is None, a scalar is returned. If an output array is specified, a reference
to out is returned.

Examples

```python
golden >>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4) >>> print x
[[1 -- 3] [-- 5 --] [7 -- 9]] >>> print x.sum() 25 >>> print x.sum(axis=1) [4 5 16] >>> print x.sum(axis=0) [8 5 12] >>> print type(x.sum(axis=0, dtype=np.int64)[0]) <type 'numpy.int64'>
```

var (axis=None, dtype=None, out=None, ddof=0)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed
for the flattened array by default, otherwise over the specified axis.

Parameters

a : array_like
Array containing numbers whose variance is desired. If a is not an array, a conversion
is attempted.

axis : int, optional
Axis along which the variance is computed. The default is to compute the variance of
the flattened array.

dtype : dtype, optional
Type to use in computing the variance. For arrays of integer type the default is float32;
for arrays of float types it is the same as the array type.
out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type is cast if necessary.

ddf : int, optional

“Delta Degrees of Freedom”: the divisor used in calculation is N − ddf, where N represents the number of elements. By default ddf is zero.

Returns

variance : ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise a reference to the output array is returned.

See Also:

std

Standard deviation

mean

Average

numpy.doc.ufuncs

Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x\text{.mean()}^2)\). \)

The mean is normally calculated as \(x\text{.sum()} / N\), where \(N = \text{len}(x)\). If, however, ddf is specified, the divisor \(N - ddf\) is used instead. In standard statistical practice, ddf=1 provides an unbiased estimator of the variance of the infinite population. ddf=0 provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a,0)
array([ 1., 1.])
>>> np.var(a,1)
array([ 0.25, 0.25])

In single precision, var() can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```
Computing the standard deviation in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
```

```
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

## Minimum/maximum

### Function version of the eponymous method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.argmax</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.argmin</code></td>
<td>Returns array of indices of the minimum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.max</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.min</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.ptp</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmax</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmin</code></td>
<td>Returns array of indices to the minimum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.max</code></td>
<td>Return maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.min</code></td>
<td>Return minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.ptp</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).</td>
</tr>
</tbody>
</table>

### `argmax` *(a, axis=None, fill_value=None)*

Function version of the eponymous method.

**Returns**

`index_array`: {integer_array}

**Parameters**

- **axis**: {None, integer}

  If None, the index is into the flattened array, otherwise along the specified axis

- **fill_value**: {var}, optional

  Value used to fill in the masked values. If None, the output of `maximum_fill_value(self._data)` is used instead.

- **out**: {None, array}, optional

  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

### Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
```

```python
>>> a.argmax(0)
array([[1, 1, 1]])
>>> a.argmax(1)
array([[2, 2]])
```
**max** (*obj*, *axis*=*None*, *out*=*None*, *fill_value*=*None*)

Return the maximum along a given axis.

**Parameters**

- **axis**: {None, int}, optional
  Axis along which to operate. By default, *axis* is None and the flattened input is used.

- **out**: array_like, optional
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- **fill_value**: {var}, optional
  Value used to fill in the masked values. If None, use the output of *maximum_fill_value*.

**Returns**

- **amax**: array_like
  New array holding the result. If *out* was specified, *out* is returned.

**See Also**:

- **maximum_fill_value**
  Returns the maximum filling value for a given datatype.

**min** (*obj*, *axis*=*None*, *out*=*None*, *fill_value*=*None*)

Return the minimum along a given axis.

**Parameters**

- **axis**: {None, int}, optional
  Axis along which to operate. By default, *axis* is None and the flattened input is used.

- **out**: array_like, optional
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- **fill_value**: {var}, optional
  Value used to fill in the masked values. If None, use the output of *minimum_fill_value*.

**Returns**

- **amin**: array_like
  New array holding the result. If *out* was specified, *out* is returned.

**See Also**:

- **minimum_fill_value**
  Returns the minimum filling value for a given datatype.

**ptp** (*obj*, *axis*=*None*, *out*=*None*, *fill_value*=*None*)

Return (maximum - minimum) along the given dimension (i.e. peak-to-peak value).

**Parameters**

- **axis**: {None, int}, optional
  Axis along which to find the peaks. If None (default) the flattened array is used.

- **out**: {None, array_like}, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

fill_value : {var}, optional

Value used to fill in the masked values.

Returns

ptp : ndarray.

A new array holding the result, unless out was specified, in which case a reference to out is returned.

argmax (axis=None, fill_value=None, out=None)

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.

Parameters

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

fill_value : {var}, optional

Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

index_array : {integer_array}

Examples

>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])

argmin (axis=None, fill_value=None, out=None)

Return array of indices to the minimum values along the given axis.

Parameters

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

fill_value : {var}, optional

Value used to fill in the masked values. If None, the output of minimum_fill_value(self._data) is used instead.

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.
Returns

{ndarray, scalar} :

If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

Examples

```python
>>> x = np.ma.array(arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> print x
[[-- --]
 [2 3]]
>>> print x.argmin(axis=0, fill_value=-1)
[0 0]
>>> print x.argmin(axis=0, fill_value=9)
[1 1]
```

max (axis=None, out=None, fill_value=None)
Return the maximum along a given axis.

Parameters

axis : {None, int}, optional
   Axis along which to operate. By default, axis is None and the flattened input is used.

out : array_like, optional
   Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

fill_value : {var}, optional
   Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns

amax : array_like
   New array holding the result. If out was specified, out is returned.

See Also:

maximum_fill_value
   Returns the maximum filling value for a given datatype.

min (axis=None, out=None, fill_value=None)
Return the minimum along a given axis.

Parameters

axis : {None, int}, optional
   Axis along which to operate. By default, axis is None and the flattened input is used.

out : array_like, optional
   Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

fill_value : {var}, optional
   Value used to fill in the masked values. If None, use the output of minimum_fill_value.
Returns

amin : array_like

New array holding the result. If out was specified, out is returned.

See Also:

minimum_fill_value
Returns the minimum filling value for a given datatype.

ptp (axis=None, out=None, fill_value=None)
Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters

axis : {None, int}, optional
Axis along which to find the peaks. If None (default) the flattened array is used.

out : {None, array_like}, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

fill_value : {var}, optional
Value used to fill in the masked values.

Returns

ptp : ndarray.
A new array holding the result, unless out was specified, in which case a reference to out is returned.

Sorting

argsort (a, axis=None, kind='quicksort', order=None, fill_value=None)
Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

Parameters

axis : int, optional
Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

fill_value : var, optional
Value used to fill the array before sorting. The default is the fill_value attribute of the input array.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
Sorting algorithm.

3.20. Masked array operations
order : list, optional

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns

index_array : ndarray, int

Array of indices that sort a along the specified axis. In other words, a[index_array] yields a sorted a.

See Also:

sort

Describes sorting algorithms used.

lexsort

Indirect stable sort with multiple keys.

ndarray.sort

Inplace sort.

Notes

See sort for notes on the different sorting algorithms.

Examples

```python
>>> a = np.ma.array([3,2,1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
              mask = [False False True],
              fill_value = 999999)
>>> a.argsort()
array([1, 0, 2])
```

sort (a, axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)

Return a sorted copy of an array.

Parameters

a : array_like

Array to be sorted.

axis : int or None, optional

Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

kind : {‘quicksort’, ‘mergesort’, ‘heapsort’}, optional

Sorting algorithm. Default is ‘quicksort’.

order : list, optional

When a is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

endwith : {True, False}, optional

Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

fill_value : {var}
Value used to fill in the masked values. If None, use the output of minimum_fill_value().

Returns

sorted_array : ndarray

Array of the same type and shape as a.

See Also:

ndarray.sort

Method to sort an array in-place.

argsort

Indirect sort.

lexsort

Indirect stable sort on multiple keys.

searchsorted

Find elements in a sorted array.

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>~n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```python
>>> a = np.array([[1, 4], [3, 1]])
>>> np.sort(a)                           # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)                # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)                   # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the order keyword to specify a field to use when sorting a structured array:

```python
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ...
            ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)    # create a structured array
>>> np.sort(a, order='height')           # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
       ('Lancelot', 1.8999999999999999, 38)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

Sort by age, then height if ages are equal:
np.sort(a, order=['age', 'height'])  # doctest: +SKIP

array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
       ('Arthur', 1.8, 41)],
dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])

argsort (axis=None, fill_value=None, kind='quicksort', order=None)

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to
fill_value.

Parameters
axis : int, optional
   Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

fill_value : var, optional
   Value used to fill the array before sorting. The default is the fill_value attribute of the input array.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
   Sorting algorithm.

order : list, optional
   When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns
index_array : ndarray, int
   Array of indices that sort a along the specified axis. In other words, a[index_array] yields a sorted a.

See Also:

sort
   Describes sorting algorithms used.

lexsort
   Indirect stable sort with multiple keys.

ndarray.sort
   Inplace sort.

Notes

See sort for notes on the different sorting algorithms.

Examples

>>> a = np.ma.array([3,2,1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
             mask = [False False True],
             fill_value = 999999)

>>> a.argsort()
array([1, 0, 2])
sort (axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)
Return a sorted copy of an array.

Parameters
   a : array_like
       Array to be sorted.
   axis : int or None, optional
       Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.
   kind : {'quicksort', 'mergesort', 'heapsort'}, optional
       Sorting algorithm. Default is 'quicksort'.
   order : list, optional
       When a is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.
   endwith : {True, False}, optional
       Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).
   fill_value : {var}
       Value used to fill in the masked values. If None, use the the output of minimum_fill_value().

Returns
   sorted_array : ndarray
       Array of the same type and shape as a.

See Also:

ndarray.sort
   Method to sort an array in-place.
argsort
   Indirect sort.
lexsort
   Indirect stable sort on multiple keys.
searchsorted
   Find elements in a sorted array.

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>-n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.
Examples

```python
golden>> a = np.array([[1,4],[3,1]])
```
```python
golden>> np.sort(a) # sort along the last axis
array([[1, 4],
       [1, 3]])
```
```python
golden>> np.sort(a, axis=None) # sort the flattened array
array([1, 1, 3, 4])
```
```python
golden>> np.sort(a, axis=0) # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the `order` keyword to specify a field to use when sorting a structured array:

```python
golden>> dtype = [ ('name', 'S10'), ('height', float), ('age', int) ]
```
```python
golden>> values = [ ('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
      ('Galahad', 1.7, 38) ]
```
```python
golden>> a = np.array(values, dtype=dtype) # create a structured array
```
```python
golden>> np.sort(a, order='height') # doctest: +SKIP
```
```python
golden>> np.sort(a, order=['age', 'height']) # doctest: +SKIP
```

Sort by age, then height if ages are equal:

```python
golden>> np.sort(a, order=['age', 'height']) # doctest: +SKIP
```
```python
golden>> np.sort(a, order=['age', 'height']) # doctest: +SKIP
```

Algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.diag(v, k)</code></td>
<td>Extract a diagonal or construct a diagonal array.</td>
</tr>
<tr>
<td><code>ma.dot(a, b[, strict])</code></td>
<td>Return the dot product of two arrays.</td>
</tr>
<tr>
<td><code>ma.identity(n[, dtype])</code></td>
<td>Return the identity array.</td>
</tr>
<tr>
<td><code>ma.inner(a, b)</code></td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td><code>ma.innerproduct(a, b)</code></td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td><code>ma.outer(a, b)</code></td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td><code>ma.outerproduct(a, b)</code></td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td><code>ma.trace(self[, offset, axis1, axis2, ...])</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>ma.transpose(a[, axes])</code></td>
<td>Permute the dimensions of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.trace</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
</tbody>
</table>

`diag(v, k=0)`

Extract a diagonal or construct a diagonal array.

This function is the equivalent of `numpy.diag` that takes masked values into account, see `numpy.diag` for details.

See Also:
**numpy.diag**
Equivalent function for ndarrays.

**dot** *(a, b, strict=False)*
Return the dot product of two arrays.

**Note:** Works only with 2-D arrays at the moment.
This function is the equivalent of **numpy.dot** that takes masked values into account, see **numpy.dot** for details.

**Parameters**
- **a, b**: ndarray
  Inputs arrays.
- **strict**: bool, optional
  Whether masked data are propagated (True) or set to 0 (False) for the computation. Default is False. Propagating the mask means that if a masked value appears in a row or column, the whole row or column is considered masked.

**See Also:**
**numpy.dot**
Equivalent function for ndarrays.

**Examples**

```python
>>> a = ma.array([[1, 2, 3], [4, 5, 6]], mask=[[1, 0, 0], [0, 0, 0]])
>>> b = ma.array([[1, 2], [3, 4], [5, 6]], mask=[[1, 0], [0, 0], [0, 0]])
>>> np.ma.dot(a, b)
masked_array(data =
[[ 21  26]
 [ 45  64]],
mask =
[[False False]
[False False]],
fill_value = 999999)
>>> np.ma.dot(a, b, strict=True)
masked_array(data =
[[ -- --]
[ -- 64]],
mask =
[[ True  True]
[ True False]],
fill_value = 999999)
```

**identity()**
Return the identity array.

The identity array is a square array with ones on the main diagonal.

**Parameters**
- **n**: int
  Number of rows (and columns) in n x n output.
- **dtype**: data-type, optional
  Data-type of the output. Defaults to float.
Returns

out : ndarray

$n \times n$ array with its main diagonal set to one, and all other elements 0.

Examples

```python
>>> np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

inner(a, b)
Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

Parameters

- a, b : array_like
  If $a$ and $b$ are nonscalar, their last dimensions of must match.

Returns

out : ndarray

$out.shape = a.shape[:-1] + b.shape[:-1]$

Raises

- ValueError
  If the last dimension of $a$ and $b$ has different size.

See Also:

tensordot
  Sum products over arbitrary axes.
dot
  Generalised matrix product, using second last dimension of $b$.

Notes

Masked values are replaced by 0.

Examples

Ordinary inner product for vectors:

```python
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```
An example where $b$ is a scalar:

```python
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

**innerproduct** $(a, b)$
Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

- $a, b$: array_like
  If $a$ and $b$ are nonscalar, their last dimensions of must match.

**Returns**

- $out$: ndarray
  $out.shape = a.shape[:-1] + b.shape[:-1]$

** Raises**

- **ValueError**: If the last dimension of $a$ and $b$ has different size.

**See Also:**

- **tensordot**: Sum products over arbitrary axes.
- **dot**: Generalised matrix product, using second last dimension of $b$.

**Notes**
Masked values are replaced by 0.

**Examples**

Ordinary inner product for vectors:

```python
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where $b$ is a scalar:

```python
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```
**outer**

(a, b)

Compute the outer product of two vectors.

Given two vectors, \(a = [a_0, a_1, \ldots, a_M]\) and \(b = [b_0, b_1, \ldots, b_N]\), the outer product \([R57]\) is:

\[
\begin{bmatrix}
  a_0*b_0 & a_0*b_1 & \cdots & a_0*b_N \\
  a_1*b_0 & a_1*b_1 & \cdots & a_1*b_N \\
  \vdots & \vdots & \ddots & \vdots \\
  a_M*b_0 & a_M*b_1 & \cdots & a_M*b_N
\end{bmatrix}
\]

**Parameters**

- a, b : array_like, shape (M,), (N,)

  First and second input vectors. Inputs are flattened if they are not already 1-dimensional.

**Returns**

- out : ndarray, shape (M, N)

  \(\text{out}[i, j] = a[i] * b[j]\)

**Notes**

Masked values are replaced by 0.

**References**

[R57]

**Examples**

Make a (very coarse) grid for computing a Mandelbrot set:

```python
g = np.mgrid[-2:2:100j,-2j:2j:100j]
```

An example using a “vector” of letters:
 NumPy Reference, Release 1.4.1

```python
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
       [b, bb, bbb],
       [c, cc, ccc]], dtype=object)
```

**outerproduct** (*a*, *b*)

Compute the outer product of two vectors.

Given two vectors, :math:`a = [a_0, a_1, \ldots, a_M]` and :math:`b = [b_0, b_1, \ldots, b_N]`, the outer product [R58] is:

\[
\begin{array}{cccc}
a_0*b_0 & a_0*b_1 & \ldots & a_0*b_N \\
a_1*b_0 & a_1*b_1 & \ldots & a_1*b_N \\
\vdots & \vdots & \ddots & \vdots \\
a_M*b_0 & a_M*b_1 & \ldots & a_M*b_N \\
\end{array}
\]

**Parameters**

*a*, *b* : array_like, shape (M,), (N,)

First and second input vectors. Inputs are flattened if they are not already 1-dimensional.

**Returns**

*out* : ndarray, shape (M, N)

[out][i, j] = a[i] * b[j]

**Notes**

Masked values are replaced by 0.

**References**

[R58]

**Examples**

Make a (very coarse) grid for computing a Mandelbrot set:

```python
>>> r1 = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> r1
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]]

>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j,  0.+2.j,  0.+2.j,  0.+2.j,  0.+2.j],
       [ 0.+1.j,  0.+1.j,  0.+1.j,  0.+1.j,  0.+1.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.-1.j,  0.-1.j,  0.-1.j,  0.-1.j,  0.-1.j],
       [ 0.-2.j,  0.-2.j,  0.-2.j,  0.-2.j,  0.-2.j]])

>>> grid = r1 + im
>>> grid
array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
       [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
       [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
       [ 0.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j],
       [ 1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j,  3.-2.j]])
```
An example using a “vector” of letters:

```python
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
       [b, bb, bbb],
       [c, cc, ccc]], dtype=object)
```

```
trace()
```

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

`numpy.trace`

equivalent function

```
transpose(a, axes=None)
```

Permute the dimensions of an array.

This function is exactly equivalent to `numpy.transpose`.

See Also:

`numpy.transpose`

Equivalent function in top-level NumPy module.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[0 1]
 [2 --]],
mask =
[[False False]
 [False  True]],
fill_value = 999999)
```

```python
>>> ma.transpose(x)
masked_array(data =
[[0 2]
 [1 --]],
mask =
[[False False]
 [False  True]],
fill_value = 999999)
```

```
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()**

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]])
```

Polynomial fit

**ma.vander** (x[, n])

Generate a Van der Monde matrix.

**ma.polyfit** (x, y, deg[, rcond, full])

Least squares polynomial fit.

**vander** (x, n=None)

Generate a Van der Monde matrix.
The columns of the output matrix are decreasing powers of the input vector. Specifically, the i-th output column is the input vector to the power of \( N - i - 1 \). Such a matrix with a geometric progression in each row is named Van Der Monde, or Vandermonde matrix, from Alexandre-Theophile Vandermonde.

**Parameters**
- \( x \): array_like
  1-D input array.
- \( N \): int, optional
  Order of (number of columns in) the output. If \( N \) is not specified, a square array is returned (\( N = \text{len}(x) \)).

**Returns**
- \( \text{out} \): ndarray
  Van der Monde matrix of order \( N \). The first column is \( x^{(N-1)} \), the second \( x^{(N-2)} \) and so forth.

**Notes**
Masked values in the input array result in rows of zeros.

**References**
[R61]

**Examples**

```python
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1, 1, 1],
       [ 4, 2, 1],
       [ 9, 3, 1],
       [25, 5, 1]])
```

```python
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1, 1, 1],
       [ 4, 2, 1],
       [ 9, 3, 1],
       [25, 5, 1]])
```

```python
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[ 1, 1, 1, 1],
       [ 8, 4, 2, 1],
       [27, 9, 3, 1],
       [125, 25, 5, 1]])
```

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```python
>>> np.linalg.det(np.vander(x))
48.000000000000043
```

```python
(5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
48
```
**polyfit** *(x, y, deg, rcond=None, full=False)*

Least squares polynomial fit.

Fit a polynomial \( p(x) = p[0] \times x^{\text{deg}} + \ldots + p[\text{deg}] \) of degree *deg* to points \((x, y)\). Returns a vector of coefficients \( p \) that minimises the squared error.

**Parameters**

- **x**: array_like, shape (M,)
  
  x-coordinates of the M sample points \((x[i], y[i])\).

- **y**: array_like, shape (M,) or (M, K)
  
  y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- **deg**: int
  
  Degree of the fitting polynomial

- **rcond**: float, optional
  
  Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \(\text{len}(x)\times\text{eps}\), where eps is the relative precision of the float type, about 2e-16 in most cases.

- **full**: bool, optional
  
  Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**Returns**

- **p**: ndarray, shape (M,) or (M, K)
  
  Polynomial coefficients, highest power first. If *y* was 2-D, the coefficients for *k*-th data set are in \(p[:, k]\).

- **residuals, rank, singular_values, rcond**: present only if full = True
  
  Residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of rcond. For more details, see `linalg.lstsq`.

**See Also:**

- **polyval**
  
  Computes polynomial values.

- **linalg.lstsq**
  
  Computes a least-squares fit.

- **scipy.interpolate.UnivariateSpline**
  
  Computes spline fits.

**Notes**

Any masked values in *x* is propagated in *y*, and vice-versa.

**References**

- [R59]
- [R60]
Examples

```python
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206, 1.69312169, -0.03968254])
```

It is convenient to use `poly1d` objects for dealing with polynomials:

```python
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```python
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```python
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
>>> plt.show()
```

```
```

```
```
Clipping and rounding

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.around()</code></td>
<td>Round an array to the given number of decimals.</td>
</tr>
<tr>
<td><code>ma.clip(a, a_min, a_max[, out])</code></td>
<td>Clip (limit) the values in an array.</td>
</tr>
<tr>
<td><code>ma.round(a[, decimals, out])</code></td>
<td>Return a copy of a, rounded to ‘decimals’ places.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.clip(a_min, a_max[, out])</code></td>
<td>Return an array whose values are limited to [a_min, a_max].</td>
</tr>
<tr>
<td><code>ma.MaskedArray.round([decimals, out])</code></td>
<td>Return an array rounded a to the given number of decimals.</td>
</tr>
</tbody>
</table>

`around()`
Round an array to the given number of decimals.

Refer to `around` for full documentation.

See Also:
`around` equivalent function

`clip(a, a_min, a_max, out=None)`
Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters
- `a` : array_like
  Array containing elements to clip.
- `a_min` : scalar or array_like
  Minimum value.
- `a_max` : scalar or array_like
  Maximum value. If `a_min` or `a_max` are array_like, then they will be broadcasted to the shape of `a`.
- `out` : ndarray, optional
  The results will be placed in this array. It may be the input array for in-place clipping. `out` must be of the right shape to hold the output. Its type is preserved.

Returns
- `clipped_array` : ndarray
  An array with the elements of `a`, but where values < `a_min` are replaced with `a_min`, and those > `a_max` with `a_max`.

See Also:
`numpy.doc.ufuncs`
Section “Output arguments”

Examples

```python
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
```

```python
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
round ($a$, $decimals=0$, $out=None$)
Return a copy of $a$, rounded to ‘decimals’ places.

When ‘decimals’ is negative, it specifies the number of positions to the left of the decimal point. The real and imaginary parts of complex numbers are rounded separately. Nothing is done if the array is not of float type and ‘decimals’ is greater than or equal to 0.

**Parameters**

- $decimals$ : int
  Number of decimals to round to. May be negative.

- $out$ : array_like
  Existing array to use for output. If not given, returns a default copy of $a$.

**Notes**
If out is given and does not have a mask attribute, the mask of $a$ is lost!

clip ()
Return an array whose values are limited to $[a_{\min}, a_{\max}]$.

Refer to numpy.clip for full documentation.

See Also:

- numpy.clip
  equivalent function

round ($decimals=0$, $out=None$)
Return an array rounded $a$ to the given number of decimals.

Refer to numpy.around for full documentation.

See Also:

- numpy.around
  equivalent function
NumPy Reference, Release 1.4.1

Miscellanea

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.allequal(a, b[, fill_value])</code></td>
<td>Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.</td>
</tr>
<tr>
<td><code>ma.allclose(a, b[, masked_equal, rtol, atol, fill_value])</code></td>
<td>Returns True if two arrays are element-wise equal within a tolerance. Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td><code>ma.apply_along_axis(func1d, axis, arr, ...)</code></td>
<td>Apply a function to 1-D slices along the given axis.</td>
</tr>
<tr>
<td><code>ma.arange([start,] stop[, step, ...])</code></td>
<td>Return evenly spaced values within a given interval.</td>
</tr>
<tr>
<td><code>ma.choose(indices, choices[, out, mode])</code></td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td><code>ma.ediff1d(arr[, to_end, to_begin])</code></td>
<td>Compute the differences between consecutive elements of an array.</td>
</tr>
<tr>
<td><code>ma.indices(dimensions[, dtype])</code></td>
<td>Return an array representing the indices of a grid.</td>
</tr>
<tr>
<td><code>ma.where(condition[, x, y])</code></td>
<td>Return a masked array with elements from x or y, depending on condition.</td>
</tr>
</tbody>
</table>

**allequal(a, b, fill_value=True)**

Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

**Parameters**

- **a, b**: array_like
  Input arrays to compare.

- **fill_value**: bool, optional
  Whether masked values in a or b are considered equal (True) or not (False).

**Returns**

- **y**: bool
  Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

**See Also:**

- `all, any, numpy.ma.allclose`

**Examples**

```python
def test_allequal_fill_value():
    a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
    print(a)
    b = array([1e10, 1e-7, -42.0])
    print(b)
    result = ma.allequal(a, b, fill_value=False)
    print(result)
```

**allclose(a, b, masked_equal=True, rtol=1.0000000000000001e-05, atol=1e-08, fill_value=None)**

Returns True if two arrays are element-wise equal within a tolerance.

**Parameters**

- **a, b**: array_like

This function is equivalent to `allclose` except that masked values are treated as equal (default) or unequal, depending on the `masked_equal` argument.
Input arrays to compare.

**masked_equal**: bool, optional

Whether masked values in `a` and `b` are considered equal (True) or not (False). They are considered equal by default.

**rtol**: float, optional

Relative tolerance. The relative difference is equal to `rtol * b`. Default is 1e-5.

**atol**: float, optional

Absolute tolerance. The absolute difference is equal to `atol`. Default is 1e-8.

**fill_value**: bool, optional

*Deprecated* - Whether masked values in `a` or `b` are considered equal (True) or not (False).

**Returns**

`y`: bool

Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

**See Also**:

`all`, `any`  
`numpy.allclose`  
the non-masked `allclose`.

**Notes**

If the following equation is element-wise True, then `allclose` returns True:

```
absolute('a' - 'b') <= ('atol' + 'rtol' * absolute('b'))
```

Return True if all elements of `a` and `b` are equal subject to given tolerances.

**Examples**

```python
>>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data = [10000000000.0 1e-07 --],
             mask = [False False True],
             fill_value = 1e+20)
>>> b = ma.array([1e10, 1e-8, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
False
```

```python
>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

Masked values are not compared directly.
>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, 42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False

apply_along_axis (func1d, axis, arr, *args, **kwargs)
Apply a function to 1-D slices along the given axis.

Execute func1d(a, *args) where func1d operates on 1-D arrays and a is a 1-D slice of arr along axis.

Parameters
func1d : function
This function should accept 1-D arrays. It is applied to 1-D slices of arr along the specified axis.
axis : integer
Axis along which arr is sliced.
arr : ndarray
Input array.
args : any
Additional arguments to func1d.

Returns
outarr : ndarray
The output array. The shape of outarr is identical to the shape of arr, except along the axis dimension, where the length of outarr is equal to the size of the return value of func1d. If func1d returns a scalar outarr will have one fewer dimensions than arr.

See Also:
apply_over_axes
Apply a function repeatedly over multiple axes.

Examples
>>> def my_func(a):
...     """Average first and last element of a 1-D array""
...     return (a[0] + a[-1]) * 0.5
... b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2., 5., 8.])

For a function that doesn’t return a scalar, the number of dimensions in outarr is the same as arr.

>>> def new_func(a):
...     """Divide elements of a by 2.""
...     return a * 0.5
... b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> np.apply_along_axis(new_func, 0, b)
array([[ 0.5,  1. ,  1.5],
       [ 2. ,  2.5,  3. ],
       [ 3.5,  4. ,  4.5]])

arange()
Return evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns a ndarray rather than a list.

Parameters
- start : number, optional
  Start of interval. The interval includes this value. The default start value is 0.
- stop : number
  End of interval. The interval does not include this value.
- step : number, optional
  Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] - out[i]. The default step size is 1. If step is specified, start must also be given.
- dtype : dtype
  The type of the output array. If dtype is not given, infer the data type from the other input arguments.

Returns
- out : ndarray
  Array of evenly spaced values.

  For floating point arguments, the length of the result is \(\text{ceil}((\text{stop} - \text{start})/\text{step})\). Because of floating point overflow, this rule may result in the last element of out being greater than stop.

See Also:
- linspace
  Evenly spaced numbers with careful handling of endpoints.
- ogrid
  Arrays of evenly spaced numbers in N-dimensions
- mgrid
  Grid-shaped arrays of evenly spaced numbers in N-dimensions

Examples

```python
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 7])
```
choose (indices, choices, out=None, mode='raise')
Use an index array to construct a new array from a set of choices.

Given an array of integers and a set of n choice arrays, this method will create a new array that merges each of the choice arrays. Where a value in a is i, the new array will have the value that choices[i] contains in the same place.

Parameters
- a : ndarray of ints
  This array must contain integers in [0, n-1], where n is the number of choices.
- choices : sequence of arrays
  Choice arrays. The index array and all of the choices should be broadcastable to the same shape.
- out : array, optional
  If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.
- mode : {'raise', 'wrap', 'clip'}, optional
  Specifies how out-of-bounds indices will behave.
  - 'raise' : raise an error
  - 'wrap' : wrap around
  - 'clip' : clip to the range

Returns
- merged_array : array

See Also:
- choose equivalent function

Examples
```python
>>> choice = np.array([[1,1,1], [2,2,2], [3,3,3]])
>>> a = np.array([2, 1, 0])
>>> np.ma.choose(a, choice)
masked_array(data = [3 2 1],
mask = False,
fill_value=999999)
```

diff1d (arr, to_end=None, to_begin=None)
Compute the differences between consecutive elements of an array.

This function is the equivalent of numpy.diff that takes masked values into account, see numpy.diff for details.

See Also:
- numpy.diff Equivalent function for ndarrays.
indices (dimensions, dtype=<type 'int'>)
    Return an array representing the indices of a grid.

    Compute an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

    Parameters
    dimensions : sequence of ints
        The shape of the grid.
    dtype : dtype, optional
        Data type of the result.

    Returns
    grid : ndarray
        The array of grid indices, grid.shape = (len(dimensions),) +
        tuple(dimensions).

    See Also:
    mgrid, meshgrid

    Notes
    The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if
    dimensions is a tuple (r0, ..., rN-1) of length N, the output shape is (N,r0,...,rN-1).

    The subarrays grid[k] contains the N-D array of indices along the k-th axis. Explicitly:

    grid[k,i0,i1,...,iN-1] = ik

    Examples

    >>> grid = np.indices((2, 3))
    >>> grid.shape
    (2,2,3)
    >>> grid[0]  # row indices
    array([[0, 0, 0],
            [1, 1, 1]])
    >>> grid[1]  # column indices
    array([[0, 1, 2],
            [0, 1, 2]])

    The indices can be used as an index into an array.

    >>> x = np.arange(20).reshape(5, 4)
    >>> row, col = np.indices((2, 3))
    >>> x[row, col]
    array([[0, 1, 2],
            [4, 5, 6]])

    Note that it would be more straightforward in the above example to extract the required elements directly with
    x[:,2, :3].

where (condition, x=None, y=None)
    Return a masked array with elements from x or y, depending on condition.
Returns a masked array, shaped like condition, where the elements are from \( x \) when \( \text{condition} \) is True, and from \( y \) otherwise. If neither \( x \) nor \( y \) are given, the function returns a tuple of indices where \( \text{condition} \) is True (the result of \( \text{condition}.\text{nonzero()} \)).

**Parameters**

- **condition**: array_like, bool
  
The condition to meet. For each True element, yield the corresponding element from \( x \), otherwise from \( y \).
- **\( x, y \)**: array_like, optional
  
  Values from which to choose. \( x \) and \( y \) need to have the same shape as condition, or be broadcastable to that shape.

**Returns**

- **out**: MaskedArray or tuple of ndarrays
  
The resulting masked array if \( x \) and \( y \) were given, otherwise the result of \( \text{condition}.\text{nonzero()} \).

**See Also:**

- **numpy.where**
  
  Equivalent function in the top-level NumPy module.

**Examples**

```python
>>> x = np.ma.array(np.arange(9.).reshape(3, 3), mask=[[0, 1, 0],
... [1, 0, 1],
... [0, 1, 0]])
>>> print x
[[ 0. -- 2.0]
 [-- 4.0 --]
[6.0 -- 8.0]]
>>> np.ma.where(x > 5) # return the indices where x > 5
(array([2, 2]), array([0, 2]))

>>> print np.ma.where(x > 5, x, -3.1416)
[[-3.1416 -- -3.1416]
 [-- -3.1416 --]
[6.0 -- 8.0]]
```

### 3.21 Numpy-specific help functions

#### 3.21.1 Finding help

```python
lookfor (what[, module, import_modules, ...])  Do a keyword search on docstrings.

lookfor (what, module=None, import_modules=True, regenerate=False, output=None)

Do a keyword search on docstrings.

A list of of objects that matched the search is displayed, sorted by relevance. All given keywords need to be found in the docstring for it to be returned as a result, but the order does not matter.

**Parameters**

- **what**: str
```
String containing words to look for.

module : str or list, optional
    Name of module(s) whose docstrings to go through.

import_modules : bool, optional
    Whether to import sub-modules in packages. Default is True.

regenerate : bool, optional
    Whether to re-generate the docstring cache. Default is False.

output : file-like, optional
    File-like object to write the output to. If omitted, use a pager.

See Also:
    source, info

Notes
Relevance is determined only roughly, by checking if the keywords occur in the function name, at the start of a
docstring, etc.

Examples

```python
>>> np.lookfor('binary representation')
Search results for 'binary representation'
------------------------------------------
numpy.binary_repr
    Return the binary representation of the input number as a string.
numpy.base_repr
    Return a string representation of a number in the given base system.
...```

3.21.2 Reading help

<table>
<thead>
<tr>
<th><code>info</code> ([object, maxwidth, ...])</th>
<th>Get help information for a function, class, or module.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>source</code> ([object[, output, mode 'w' at 0x323070&gt;])</td>
<td>Print or write to a file the source code for a Numpy object.</td>
</tr>
</tbody>
</table>

`info` (object=None, maxwidth=76, output=<open file '<stdout>', mode 'w' at 0x323070>, toplevel='numpy')
Get help information for a function, class, or module.

Parameters

- **object** : object or str, optional
  Input object or name to get information about. If `object` is a numpy object, its docstring
  is given. If it is a string, available modules are searched for matching objects. If None,
  information about `info` itself is returned.

- **maxwidth** : int, optional
  Printing width.

- **output** : file like object, optional
  File like object that the output is written to, default is `stdout`. The object has to be
  opened in 'w' or 'a' mode.

- **toplevel** : str, optional
Start search at this level.

See Also:

source, lookfor

Notes

When used interactively with an object, np.info(obj) is equivalent to help(obj) on the Python prompt or obj? on the IPython prompt.

Examples

```python
>>> np.info(np.polyval) # doctest: +SKIP
polyval(p, x)
   Evaluate the polynomial p at x.
   ...
```

When using a string for object it is possible to get multiple results.

```python
>>> np.info('fft') # doctest: +SKIP
*** Found in numpy ***
Core FFT routines
   ...
   *** Found in numpy.fft ***
   fft(a, n=None, axis=-1)
   ...
   *** Repeat reference found in numpy.fft.fftpack ***
   *** Total of 3 references found. ***
```

source(object, output=<open file '<stdout>', mode 'w' at 0x323070>)

Print or write to a file the source code for a Numpy object.

The source code is only returned for objects written in Python. Many functions and classes are defined in C and will therefore not return useful information.

Parameters

object : numpy object

Input object. This can be any object (function, class, module, ...).

output : file object, optional

If output not supplied then source code is printed to screen (sys.stdout). File object must be created with either write ‘w’ or append ‘a’ modes.

See Also:

lookfor, info

Examples

```python
>>> np.source(np.interp)
In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
def interp(x, xp, fp, left=None, right=None):
    """... (full docstring printed)""
    if isinstance(x, (float, int, number)):
        return compiled_interp([x], xp, fp, left, right).item()
    else:
        return compiled_interp(x, xp, fp, left, right)
```
The source code is only returned for objects written in Python.

```python
>>> np.source(np.array)
Not available for this object.
```

### 3.22 Miscellaneous routines

#### 3.22.1 Buffer objects

<table>
<thead>
<tr>
<th><code>getbuffer</code> (obj [,offset[, size]])</th>
<th>Create a buffer object from the given object referencing a slice of length size starting at offset.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newbuffer</code> (size)</td>
<td>Return a new uninitialized buffer object of size bytes</td>
</tr>
</tbody>
</table>

**getbuffer()**

Create a buffer object from the given object referencing a slice of length size starting at offset.

Default is the entire buffer. A read-write buffer is attempted followed by a read-only buffer.

**Parameters**

- `obj` : object
- `offset` : int, optional
- `size` : int, optional

**Returns**

- `buffer_obj` : buffer

**Examples**

```python
>>> buf = np.getbuffer(np.ones(5), 1, 3)
>>> len(buf)
3
>>> buf[0]
'\x00'
>>> buf
<read-write buffer for 0x8af1e70, size 3, offset 1 at 0x8ba4ec0>
```

**newbuffer()**

Return a new uninitialized buffer object of size bytes

#### 3.22.2 Performance tuning

<table>
<thead>
<tr>
<th><code>alterdot</code>()</th>
<th>Change <code>dot</code>, <code>vdot</code>, and <code>innerproduct</code> to use accelerated BLAS functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>restoredot</code>()</td>
<td>Restore <code>dot</code>, <code>vdot</code>, and <code>innerproduct</code> to the default non-BLAS implementations.</td>
</tr>
<tr>
<td><code>setbufsize</code>(size)</td>
<td>Set the size of the buffer used in ufuncs.</td>
</tr>
<tr>
<td><code>getbufsize</code>()</td>
<td>Return the size of the buffer used in ufuncs.</td>
</tr>
</tbody>
</table>

**alterdot()**

Change `dot`, `vdot`, and `innerproduct` to use accelerated BLAS functions.

Typically, as a user of Numpy, you do not explicitly call this function. If Numpy is built with an accelerated BLAS, this function is automatically called when Numpy is imported.

When Numpy is built with an accelerated BLAS like ATLAS, these functions are replaced to make use of the faster implementations. The faster implementations only affect float32, float64, complex64, and complex128.
arrays. Furthermore, the BLAS API only includes matrix-matrix, matrix-vector, and vector-vector products. Products of arrays with larger dimensionalities use the built in functions and are not accelerated.

See Also:

restoredot

restoredot undoes the effects of alterdot.

restoredot()

Restore dot, vdot, and innerproduct to the default non-BLAS implementations.

Typically, the user will only need to call this when troubleshooting and installation problem, reproducing the conditions of a build without an accelerated BLAS, or when being very careful about benchmarking linear algebra operations.

See Also:

alterdot

alterdot undoes the effects of alterdot.

setbufsize(size)

Set the size of the buffer used in ufuncs.

Parameters

size : int

Size of buffer.

getbufsize()

Return the size of the buffer used in ufuncs.

3.23 Test Support (numpy.testing)

Common test support for all numpy test scripts.

This single module should provide all the common functionality for numpy tests in a single location, so that test scripts can just import it and work right away.
### 3.24 Asserts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert_almost_equal</td>
<td>Raise an assertion if two items are not equal up to desired precision.</td>
</tr>
<tr>
<td>assert_approx_equal</td>
<td>Raise an assertion if two items are not equal up to significant digits.</td>
</tr>
<tr>
<td>assert_array_almost_equal</td>
<td>Raise an assertion if two objects are not equal up to desired precision.</td>
</tr>
<tr>
<td>assert_array_equal</td>
<td>Raise an assertion if two array_like objects are not equal.</td>
</tr>
<tr>
<td>assert_array_less</td>
<td>Raise an assertion if two array_like objects are not ordered by less than.</td>
</tr>
<tr>
<td>assert_equal</td>
<td>Raise an assertion if two objects are not equal.</td>
</tr>
<tr>
<td>assert_raises</td>
<td>Fail unless an exception of class exception_class is thrown by callable when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.</td>
</tr>
<tr>
<td>assert_warns</td>
<td>Fail unless a warning of class warning_class is thrown by callable when invoked with arguments args and keyword arguments kwargs.</td>
</tr>
<tr>
<td>assert_string_equal</td>
<td>Test if two strings are equal.</td>
</tr>
</tbody>
</table>

#### assert_almost_equal

Raise an assertion if two items are not equal up to desired precision.

The test is equivalent to \( \text{abs(desired-actual)} < 0.5 * 10^{-\text{decimal}} \)

Given two objects (numbers or ndarrays), check that all elements of these objects are almost equal. An exception is raised at conflicting values. For ndarrays this delegates to assert_array_almost_equal

**Parameters**

- **actual**: number or ndarray  
  The object to check.
- **desired**: number or ndarray  
  The expected object.
- **decimal**: integer (decimal=7)  
  desired precision
- **err_msg**: string  
  The error message to be printed in case of failure.
- **verbose**: bool  
  If True, the conflicting values are appended to the error message.

**Raises**

- **AssertionError**:  
  If actual and desired are not equal up to specified precision.

**See Also**
assert_array_almost_equal
   compares array_like objects

assert_equal
   tests objects for equality

Examples

>>> npt.assert_almost_equal(2.3333333333333, 2.33333334)
>>> npt.assert_almost_equal(2.3333333333333, 2.33333334, decimal=10)
...<type 'exceptions.AssertionError'>:
Items are not equal:
   ACTUAL: 2.3333333333333002
   DESIRED: 2.3333333399999998

>>> npt.assert_almost_equal(np.array([1.0, 2.3333333333333]),
np.array([1.0, 2.33333334]), decimal=9)
...
<type 'exceptions.AssertionError'>:
Arrays are not almost equal
<BLANKLINE>
   (mismatch 50.0%)
   x: array([ 1. , 2.33333333])
   y: array([ 1. , 2.33333334])

assert_approx_equal (actual, desired, significant=7, err_msg=", verbose=True)
Raise an assertion if two items are not equal up to significant digits.

Given two numbers, check that they are approximately equal. Approximately equal is defined as the number of significant digits that agree.

Parameters
   actual : number
      The object to check.
   desired : number
      The expected object.
   significant : integer (significant=7)
      desired precision
   err_msg : string
      The error message to be printed in case of failure.
   verbose : bool
      If True, the conflicting values are appended to the error message.

Raises
   AssertionError 
      If actual and desired are not equal up to specified precision.

See Also:

assert_almost_equal
   compares objects by decimals
assert_array_almost_equal
   compares array_like objects by decimals

assert_equal
   tests objects for equality

Examples

>>> np.testing.assert_approx_equal(0.12345677777777e-20, 0.1234567e-20)

>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345671e-20,
   significant=8)

>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345672e-20,
   significant=8)
...
<type 'exceptions.AssertionError'>:
Items are not equal to 8 significant digits:
   ACTUAL: 1.234567e-021
   DESIRED: 1.2345672000000001e-021

the evaluated condition that raises the exception is

>>> abs(0.12345670e-20/1e-21 - 0.12345672e-20/1e-21) >= 10**-(8-1)
True

assert_array_almost_equal(x, y, decimal=6, err_msg=", verbose=True)
Raise an assertion if two objects are not equal up to desired precision.

The test verifies identical shapes and verifies values with abs(desired-actual) < 0.5 * 10**(-decimal)

Given two array_like objects, check that the shape is equal and all elements of these objects are almost equal.
An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

Parameters
   x : array_like
      The actual object to check.
   y : array_like
      The desired, expected object.
   decimal : integer (decimal=6)
      desired precision
   err_msg : string
      The error message to be printed in case of failure.
   verbose : bool
      If True, the conflicting values are appended to the error message.

Raises
   AssertionError :
      If actual and desired are not equal up to specified precision.

See Also:

 assert_almost_equal
   simple version for comparing numbers
assert_array_equal
tests objects for equality

Examples

the first assert does not raise an exception

```python
>>> np.testing.assert_array_almost_equal([1.0, 2.333, np.nan],
                                    [1.0, 2.333, np.nan])
```

```python
>>> np.testing.assert_array_almost_equal([1.0, 2.33333, np.nan],
                                    [1.0, 2.33339, np.nan], decimal=5)
...<type 'exceptions.AssertionError'>:
AssertionError:
Arrays are not almost equal
<BLANKLINE>
(mismatch 50.0%)
x: array([ 1. , 2.33333, NaN])
y: array([ 1. , 2.33339, NaN])
```

```python
>>> np.testing.assert_array_almost_equal([1.0, 2.33333, np.nan],
                                    [1.0, 2.333, 5], decimal=5)
<type 'exceptions.ValueError'>:
ValueError:
Arrays are not almost equal
x: array([ 1. , 2.33333, NaN])
y: array([ 1. , 2.3333, 5. ])
```

assert_array_equal(x, y, err_msg=", verbose=True)
Raise an assertion if two array_like objects are not equal.

Given two array_like objects, check that the shape is equal and all elements of these objects are equal. An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

The usual caution for verifying equality with floating point numbers is advised.

Parameters

- x : array_like
  The actual object to check.

- y : array_like
  The desired, expected object.

- err_msg : string
  The error message to be printed in case of failure.

- verbose : bool
  If True, the conflicting values are appended to the error message.

Raises

- AssertionError
  If actual and desired objects are not equal.

See Also:
assert_array_almost_equal
test objects for equality up to precision

assert_equal
tests objects for equality

Examples
the first assert does not raise an exception

>>> np.testing.assert_array_equal([1.0,2.33333, np.nan],
                                [np.exp(0),2.33333, np.nan])

assert fails with numerical inprecision with floats

>>> np.testing.assert_array_equal([1.0, np.pi, np.nan],
                                [1, np.sqrt(np.pi)**2, np.nan])
...
<type 'exceptions.ValueError'>:
AssertionError:
Arrays are not equal
<BLANKLINE>
(match 50.0%)
    x: array([ 1.   , 3.14159265, NaN])
    y: array([ 1.   , 3.14159265, NaN])

use assert_array_almost_equal for these cases instead

>>> np.testing.assert_array_almost_equal([1.0, np.pi, np.nan],
                        [1, np.sqrt(np.pi)**2, np.nan], decimal=15)

assert_array_less (x, y, err_msg="", verbose=True)
Raise an assertion if two array_like objects are not ordered by less than.

Given two array_like objects, check that the shape is equal and all elements of the first object are strictly smaller than those of the second object. An exception is raised at shape mismatch or incorrectly ordered values. Shape mismatch does not raise if an object has zero dimension. In contrast to the standard usage in numpy, NaNs are compared, no assertion is raised if both objects have NaNs in the same positions.

Parameters
x : array_like
    The smaller object to check.
y : array_like
    The larger object to compare.
err_msg : string
    The error message to be printed in case of failure.
verbose : bool
    If True, the conflicting values are appended to the error message.

Raises
    AssertionError :
    If actual and desired objects are not equal.

See Also:
assert_array_equal
tests objects for equality

assert_array_almost_equal
tests objects for equality up to precision

Examples

```python
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1.1, 2.0, np.nan])
...<type 'exceptions.ValueError'>:
Arrays are not less-ordered
  (mismatch 50.0%)
  x: array([ 1., 1., NaN])
  y: array([ 1., 2., NaN])
```

```python
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1, 2.0, np.nan])
...<type 'exceptions.ValueError'>:
Arrays are not less-ordered
  (mismatch 50.0%)
  x: array([ 1., 1., NaN])
  y: array([ 1., 2., NaN])
```

```python
>>> np.testing.assert_array_less([1.0, 4.0], 3)
...<type 'exceptions.ValueError'>:
Arrays are not less-ordered
  (mismatch 50.0%)
  x: array([ 1., 4.])
  y: array(3)
```

```python
>>> np.testing.assert_array_less([1.0, 2.0, 3.0], [4])
...<type 'exceptions.ValueError'>:
Arrays are not less-ordered
  (shapes (3,), (1,) mismatch)
  x: array([ 1., 2., 3.])
  y: array([4])
```

assert_equal(actual, desired, err_msg=None, verbose=True)
Raise an assertion if two objects are not equal.

Given two objects (lists, tuples, dictionaries or numpy arrays), check that all elements of these objects are equal. An exception is raised at the first conflicting values.

Parameters

- **actual**: list, tuple, dict or ndarray
  The object to check.

- **desired**: list, tuple, dict or ndarray
  The expected object.

- **err_msg**: string
  The error message to be printed in case of failure.

- **verbose**: bool
  If True, the conflicting values are appended to the error message.

Raises

- **AssertionError**
  If actual and desired are not equal.
Examples

>>> np.testing.assert_equal([4, 5], [4, 6])
...
<type 'exceptions.AssertionError'>:
Items are not equal:
 item=1
     ACTUAL: 5
     DESIRED: 6

assert_raises (exception_class, callable, *args, **kwargs)
Fail unless an exception of class exception_class is thrown by callable when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

assert_warns (warning_class, func, *args, **kwargs)
Fail unless a warning of class warning_class is thrown by callable when invoked with arguments args and keyword arguments kwargs.
If a different type of warning is thrown, it will not be caught, and the test case will be deemed to have suffered an error.

assert_string_equal (actual, desired)
Test if two strings are equal.
If the given strings are equal, assert_string_equal does nothing. If they are not equal, an AssertionError is raised, and the diff between the strings is shown.

Parameters

    actual : str
The string to test for equality against the expected string.

    desired : str
The expected string.

Examples

>>> np.testing.assert_string_equal('abc', 'abc')
>>> np.testing.assert_string_equal('abc', 'abcd')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...  # AssertionError
AssertionError: Differences in strings:
- abc
   + abcd

3.24.1 Decorators

decorators.deprecated ([conditional])
Filter deprecation warnings while running the test suite.
Make function raise KnownFailureTest exception if given condition is true.
Signals to nose that this function is or is not a test.
Make function raise SkipTest exception if a given condition is true.
Label a test as ‘slow’.
Apply a decorator to all methods in a class matching a regular expression.
**deprecated** *(conditional=True)*

Filter deprecation warnings while running the test suite.

This decorator can be used to filter DeprecationWarning’s, to avoid printing them during the test suite run, while checking that the test actually raises a DeprecationWarning.

**Parameters**

- **conditional**: bool or callable, optional
  
  Flag to determine whether to mark test as deprecated or not. If the condition is a callable, it is used at runtime to dynamically make the decision. Default is True.

**Returns**

- **decorator**: function
  
  The `deprecated` decorator itself.

**Notes**

New in version 1.4.0.

**knownfailureif** *(fail_condition, msg=None)*

Make function raise KnownFailureTest exception if given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

**Parameters**

- **fail_condition**: bool or callable
  
  Flag to determine whether to mark the decorated test as a known failure (if True) or not (if False).

- **msg**: str, optional
  
  Message to give on raising a KnownFailureTest exception. Default is None.

**Returns**

- **decorator**: function
  
  Decorator, which, when applied to a function, causes SkipTest to be raised when `skip_condition` is True, and the function to be called normally otherwise.

**Notes**

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

**setastest** *(tf=True)*

Signals to nose that this function is or is not a test.

**Parameters**

- **tf**: bool
  
  If True, specifies that the decorated callable is a test. If False, specifies that the decorated callable is not a test. Default is True.

**Notes**

This decorator can’t use the nose namespace, because it can be called from a non-test module. See also `istest` and `nottest` in `nose.tools`. 
Examples

`setastest` can be used in the following way:

```python
from numpy.testing.decorators import setastest

@setastest(False)
def func_with_test_in_name(arg1, arg2):
    pass
```

`skipif` *(skip_condition, msg=None)*

Make function raise SkipTest exception if a given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

**Parameters**

- **skip_condition**: bool or callable
  Flag to determine whether to skip the decorated test.

- **msg**: str, optional
  Message to give on raising a SkipTest exception. Default is None.

**Returns**

- **decorator**: function
  Decorator which, when applied to a function, causes SkipTest to be raised when *skip_condition* is True, and the function to be called normally otherwise.

**Notes**

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

`snow(t)`

Label a test as ‘slow’.

The exact definition of a slow test is obviously both subjective and hardware-dependent, but in general any individual test that requires more than a second or two should be labeled as slow (the whole suite consists of thousands of tests, so even a second is significant).

**Parameters**

- **t**: callable
  The test to label as slow.

**Returns**

- **t**: callable
  The decorated test t.

**Examples**

The `numpy.testing` module includes import decorators as dec. A test can be decorated as slow like this:

```python
from numpy.testing import *

@dec.slow
def test_big(self):
    print ‘Big, slow test’
```
**decorate_methods** *(cls, decorator, testmatch=None)*

Apply a decorator to all methods in a class matching a regular expression.

The given decorator is applied to all public methods of `cls` that are matched by the regular expression `testmatch` (`testmatch.search(methodname)`). Methods that are private, i.e. start with an underscore, are ignored.

**Parameters**

- **cls**: class
  Class whose methods to decorate.

- **decorator**: function
  Decorator to apply to methods

- **testmatch**: compiled regexp or str, optional
  The regular expression. Default value is None, in which case the nose default (`re.compile(r'(?::^|\b_.\%s-)]Ttest' % os.sep)`) is used. If `testmatch` is a string, it is compiled to a regular expression first.

### 3.24.2 Test Running

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tester</strong></td>
<td></td>
</tr>
<tr>
<td><code>run_module_suite()</code></td>
<td>Nose test runner.</td>
</tr>
<tr>
<td><code>rundocs()</code></td>
<td>Run doctests found in the given file.</td>
</tr>
</tbody>
</table>

**Tester**

alias of `NoseTester`

**run_module_suite** *(file_to_run=None)*

**rundocs** *(filename=None, raise_on_error=True)*

Run doctests found in the given file.

By default `rundocs` raises an `AssertionError` on failure.

**Parameters**

- **filename**: str
  The path to the file for which the doctests are run.

- **raise_on_error**: bool
  Whether to raise an `AssertionError` when a doctest fails. Default is True.

**Notes**

The doctests can be run by the user/developer by adding the `doctests` argument to the `test()` call. For example, to run all tests (including doctests) for `numpy.lib`:

```python
>>> np.lib.test(doctests=True)
```
3.25 Mathematical functions with automatic domain (numpy.emath)

Note: numpy.emath is a preferred alias for numpy.lib.scimath, available after numpy is imported. Wrapper functions to more user-friendly calling of certain math functions whose output data-type is different than the input data-type in certain domains of the input.

For example, for functions like log() with branch cuts, the versions in this module provide the mathematically valid answers in the complex plane:

```python
>>> import math
>>> from numpy import lib as scimath
>>> scimath.log(-math.exp(1)) == (1j+math.pi)
True
```

Similarly, sqrt(), other base logarithms, power() and trig functions are correctly handled. See their respective docstrings for specific examples.

3.26 Matrix library (numpy.matlib)

This module contains all functions in the numpy namespace, with the following replacement functions that return matrices instead of ndarrays.

3.27 Optionally Scipy-accelerated routines (numpy.dual)

Aliases for functions which may be accelerated by Scipy.

Scipy can be built to use accelerated or otherwise improved libraries for FFTs, linear algebra, and special functions. This module allows developers to transparently support these accelerated functions when scipy is available but still support users who have only installed Numpy.

3.27.1 Linear algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky(a)</td>
<td>Cholesky decomposition.</td>
</tr>
<tr>
<td>det(a)</td>
<td>Compute the determinant of an array.</td>
</tr>
<tr>
<td>eig(a)</td>
<td>Compute the eigenvalues and right eigenvectors of a square array.</td>
</tr>
<tr>
<td>eigh(a[, UPLO])</td>
<td>Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.</td>
</tr>
<tr>
<td>eigvals(a)</td>
<td>Compute the eigenvalues of a general matrix.</td>
</tr>
<tr>
<td>eighavalsh(a[, UPLO])</td>
<td>Compute the eigenvalues of a Hermitian or real symmetric matrix.</td>
</tr>
<tr>
<td>inv(a)</td>
<td>Compute the (multiplicative) inverse of a matrix.</td>
</tr>
<tr>
<td>lstsq(a, b[, rcond])</td>
<td>Return the least-squares solution to a linear matrix equation.</td>
</tr>
<tr>
<td>norm(x[, ord])</td>
<td>Matrix or vector norm.</td>
</tr>
<tr>
<td>pinv(a[, rcond])</td>
<td>Compute the (Moore-Penrose) pseudo-inverse of a matrix.</td>
</tr>
<tr>
<td>solve(a, b)</td>
<td>Solve a linear matrix equation, or system of linear scalar equations.</td>
</tr>
<tr>
<td>svd(a[, full_matrices, compute_uv])</td>
<td>Singular Value Decomposition.</td>
</tr>
</tbody>
</table>
3.27.2 FFT

<table>
<thead>
<tr>
<th>fft (a[, n, axis])</th>
<th>Compute the one-dimensional discrete Fourier Transform.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft2 (a[, s, axes, -1])</td>
<td>Compute the 2-dimensional discrete Fourier Transform</td>
</tr>
<tr>
<td>fftn (a[, s, axes])</td>
<td>Compute the N-dimensional discrete Fourier Transform.</td>
</tr>
<tr>
<td>ifft (a[, n, axis])</td>
<td>Compute the one-dimensional inverse discrete Fourier Transform.</td>
</tr>
<tr>
<td>ifft2 (a[, s, axes, -1])</td>
<td>Compute the 2-dimensional inverse discrete Fourier Transform.</td>
</tr>
<tr>
<td>ifftn (a[, s, axes])</td>
<td>Compute the N-dimensional inverse discrete Fourier Transform.</td>
</tr>
</tbody>
</table>

3.27.3 Other

| i0 (x) | Modified Bessel function of the first kind, order 0. |

3.28 Numarray compatibility (numpy.numarray)

3.29 Old Numeric compatibility (numpy.oldnumeric)

3.30 C-Types Foreign Function Interface (numpy.ctypeslib)

**as_array**(obj)
Create a numpy array from a ctypes array. The numpy array shares the memory with the ctypes object.

**as_ctypes**(obj)
Create and return a ctypes object from a numpy array. Actually anything that exposes the __array_interface__ is accepted.

**ctypes_load_library**(args, kwds)
*ctypes_load_library* is deprecated, use *load_library* instead!

**load_library**(libname, loader_path)

**ndpointer**(dtype=None, ndim=None, shape=None, flags=None)
Array-checking restype/argtypes.

An ndpointer instance is used to describe an ndarray in restypes and argtypes specifications. This approach is more flexible than using, for example, *POINTER(c_double)*, since several restrictions can be specified, which are verified upon calling the ctypes function. These include data type, number of dimensions, shape and flags. If a given array does not satisfy the specified restrictions, a *TypeError* is raised.

**Parameters**

- **dtype**: data-type, optional
  - Array data-type.

- **ndim**: int, optional
  - Number of array dimensions.

- **shape**: tuple of ints, optional
  - Array shape.

- **flags**: str or tuple of str
  - Array flags; may be one or more of:
NumPy Reference, Release 1.4.1

- C_CONTIGUOUS / C / CONTIGUOUS
- F_CONTIGUOUS / F / FORTRAN
- OWNDATA / O
- WRITEABLE / W
- ALIGNED / A
- UPDATEIFCOPY / U

**Returns**

- `klass` : ndpointer type object

  A type object, which is an `_ndtptr` instance containing dtype, ndim, shape and flags information.

**Raises**

- `TypeError` :

  If a given array does not satisfy the specified restrictions.

**Examples**

```python
>>> clib.somefunc.argtypes = [np.ctypeslib.ndpointer(dtype=np.float64,
... ndim=1,
... flags='C_CONTIGUOUS')]
>>> clib.somefunc(np.array([1, 2, 3], dtype=np.float64))
```

### 3.31 String operations

This module provides a set of vectorized string operations for arrays of type `numpy.string_` or `numpy.unicode_`. All of them are based on the string methods in the Python standard library.
### 3.31.1 String operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x1, x2)</code></td>
<td>Return ((x1 + x2)), that is string concatenation, element-wise for a pair of array_likes of str or unicode.</td>
</tr>
<tr>
<td><code>multiply(a, i)</code></td>
<td>Return ((a * i)), that is string multiple concatenation, element-wise.</td>
</tr>
<tr>
<td><code>mod(a, values)</code></td>
<td>Return ((a % i)), that is pre-Python 2.6 string formatting (interpolation), element-wise for a pair of array_likes of str or unicode.</td>
</tr>
<tr>
<td><code>capitalize(a)</code></td>
<td>Return a copy of (a) with only the first character of each element capitalized.</td>
</tr>
<tr>
<td><code>center(a, width[, fillchar])</code></td>
<td>Return a copy of (a) with its elements centered in a string of length (width).</td>
</tr>
<tr>
<td><code>decode(a[, encoding, errors])</code></td>
<td>Calls <code>str.decode</code> element-wise.</td>
</tr>
<tr>
<td><code>encode(a[, encoding, errors])</code></td>
<td>Calls <code>str.encode</code> element-wise.</td>
</tr>
<tr>
<td><code>join(sep, seq)</code></td>
<td>Return a string which is the concatenation of the strings in the sequence (seq).</td>
</tr>
<tr>
<td><code>ljust(a, width[, fillchar])</code></td>
<td>Return an array with the elements of (a) left-justified in a string of length (width).</td>
</tr>
<tr>
<td><code>lower(a)</code></td>
<td>Return an array with the elements of (a) converted to lowercase.</td>
</tr>
<tr>
<td><code>lstrip(a[, chars])</code></td>
<td>For each element in (a), return a copy with the leading characters removed.</td>
</tr>
<tr>
<td><code>partition(a, sep)</code></td>
<td>Partition each element in (a) around (sep).</td>
</tr>
<tr>
<td><code>replace(a, old, new[, count])</code></td>
<td>For each element in (a), return a copy of the string with all occurrences of substring (old) replaced by (new).</td>
</tr>
<tr>
<td><code>rjust(a, width[, fillchar])</code></td>
<td>Return an array with the elements of (a) right-justified in a string of length (width).</td>
</tr>
<tr>
<td><code>rpartition(a, sep)</code></td>
<td>Partition each element in (a) around (sep).</td>
</tr>
<tr>
<td><code>rsplit(a[, sep, maxsplit])</code></td>
<td>For each element in (a), return a list of the words in the string, using (sep) as the delimiter string.</td>
</tr>
<tr>
<td><code>rstrip(a[, chars])</code></td>
<td>For each element in (a), return a copy with the trailing characters removed.</td>
</tr>
<tr>
<td><code>split(a[, sep, maxsplit])</code></td>
<td>For each element in (a), return a list of the words in the string, using (sep) as the delimiter string.</td>
</tr>
<tr>
<td><code>splitlines(a[, keepends])</code></td>
<td>For each element in (a), return a list of the lines in the element, breaking at line boundaries.</td>
</tr>
<tr>
<td><code>strip(a[, chars])</code></td>
<td>For each element in (a), return a copy with the leading and trailing characters removed.</td>
</tr>
<tr>
<td><code>swapcase(a)</code></td>
<td>For each element in (a), return a copy of the string with uppercase characters converted to lowercase and vice versa.</td>
</tr>
<tr>
<td><code>title(a)</code></td>
<td>For each element in (a), return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.</td>
</tr>
<tr>
<td><code>translate(a, table[, deletechars])</code></td>
<td>For each element in (a), return a copy of the string where all characters occurring in the optional argument <code>deletechars</code> are removed, and the remaining characters have been mapped through the given translation table.</td>
</tr>
<tr>
<td><code>upper(a)</code></td>
<td>Return an array with the elements of (a) converted to uppercase.</td>
</tr>
<tr>
<td><code>zfill(a, width)</code></td>
<td>Return the numeric string left-filled with zeros in a string of length (width).</td>
</tr>
</tbody>
</table>

#### add \((x1, x2)\)

Return \((x1 + x2)\), that is string concatenation, element-wise for a pair of array_likes of str or unicode.

**Parameters**

- \(x1\) : array_like of str or unicode
- \(x2\) : array_like of str or unicode

**Returns**

- \(out\) : ndarray

Output array of `string_` or `unicode_`, depending on input types

#### multiply \((a, i)\)

Return \((a * i)\), that is string multiple concatenation, element-wise.

Values in \(i\) of less than 0 are treated as 0 (which yields an empty string).
Parameters
  a : array_like of str or unicode
  i : array_like of ints

Returns
  out : ndarray
      Output array of str or unicode, depending on input types

mod(a, values)
    Return (a % i), that is pre-Python 2.6 string formatting (interpolation), element-wise for a pair of array_likes of
    str or unicode.

    Parameters
     a : array_like of str or unicode
        values : array_like of values
            These values will be element-wise interpolated into the string.

    Returns
     out : ndarray
            Output array of str or unicode, depending on input types

See Also:
str.__mod__
capitalize(a)
    Return a copy of a with only the first character of each element capitalized.
    Calls str.capitalize element-wise.
    For 8-bit strings, this method is locale-dependent.

    Parameters
      a : array_like of str or unicode

    Returns
      out : ndarray
          Output array of str or unicode, depending on input types

See Also:
str.capitalize

Examples

>>> c = np.array([‘alb2’, ’lb2a’, ’b2al’, ’2alb’],’S4’); c
array([’alb2’, ’lb2a’, ’b2al’, ’2alb’],
dtype=’|S4’)

>>> np.char.capitalize(c)
array([’A1b2’, ’1b2a’, ’B2a1’, ’2a1b’],
dtype=’|S4’)
center(a, width, fillchar=’ ‘)
    Return a copy of a with its elements centered in a string of length width.
    Calls str.center element-wise.

    Parameters
      a : array_like of str or unicode
width : int
    The length of the resulting strings

fillchar : str or unicode, optional
    The padding character to use (default is space).

Returns
out : ndarray
    Output array of str or unicode, depending on input types

See Also:
str.center

decode (a, encoding=None, errors=None)
    Calls str.decode element-wise.
    The set of available codecs comes from the Python standard library, and may be extended at runtime. For more
    information, see the codecs module.

Parameters
a : array_like of str or unicode
    encoding : str, optional
        The name of an encoding
    errors : str, optional
        Specifies how to handle encoding errors

Returns
out : ndarray

See Also:
str.decode

Notes
The type of the result will depend on the encoding specified.

Examples
>>> c = np.array(['aAaAaA', ' aA ', 'abBABba'])
>>> c
array(['aAaAaA', ' aA', 'abBABba'],
      dtype='|S7')
>>> np.char.encode(c, encoding='cp037')
array(['\x81\xc1\x81\xc1\x81\xc1', '@@\x81\xc1@@',
       '\x81\x82\xc2\xc1\xc2\x82\x81'],
      dtype='|S7')

encode (a, encoding=None, errors=None)
    Calls str.encode element-wise.
    The set of available codecs comes from the Python standard library, and may be extended at runtime. For more
    information, see the codecs module.

Parameters
a : array_like of str or unicode
encoding : str, optional
The name of an encoding

errors : str, optional
Specifies how to handle encoding errors

Returns
out : ndarray

See Also:
str.encode

Notes
The type of the result will depend on the encoding specified.

join(sep, seq)
Return a string which is the concatenation of the strings in the sequence seq.
Calls str.join element-wise.

Parameters
sep : array_like of str or unicode
seq : array_like of str or unicode

Returns
out : ndarray
Output array of str or unicode, depending on input types

See Also:
str.join

ljust(a, width, fillchar=' ')
Return an array with the elements of a left-justified in a string of length width.
Calls str.ljust element-wise.

Parameters
a : array_like of str or unicode
width : int
The length of the resulting strings
fillchar : str or unicode, optional
The character to use for padding

Returns
out : ndarray
Output array of str or unicode, depending on input type

See Also:
str.ljust

lower(a)
Return an array with the elements of a converted to lowercase.
Call str.lower element-wise.
For 8-bit strings, this method is locale-dependent.

**Parameters**

- `a` : array-like of str or unicode

**Returns**

- `out` : ndarray, str or unicode

  Output array of str or unicode, depending on input type

**See Also:**

- `str.lower`

**Examples**

```python
>>> c = np.array([\'A1B C\', \'1BCA\', \'BCA1\']); c
array([\'A1B C\', \'1BCA\', \'BCA1\'],
      dtype='|S5')
>>> np.char.lower(c)
array([\'a1b c\', \'1bca\', \'bca1\'],
      dtype='|S5')
```

**lstrip** *(a, chars=None)*

For each element in `a`, return a copy with the leading characters removed.

Calls `str.lstrip` element-wise.

**Parameters**

- `a` : array-like of str or unicode

- `chars` : str or unicode, optional

  The `chars` argument is a string specifying the set of characters to be removed. If omitted or None, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix; rather, all combinations of its values are stripped.

**Returns**

- `out` : ndarray, str or unicode

  Output array of str or unicode, depending on input type

**See Also:**

- `str.lstrip`

**Examples**

```python
>>> c = np.array([\'aAaAaA\', \' aA\', \'abBABba\'])
>>> c
array([\'aAaAaA\', \' aA\', \'abBABba\'],
      dtype='|S7')
>>> np.char.lstrip(c, \'a\')  # \'a\' unstripped from c[1] because whitespace leading
array([\'AaAaA\', \' aA\', \'bBABba\'],
      dtype='|S6')
>>> np.char.lstrip(c, \'A\')  # leaves c unchanged
array([\'aAaAaA\', \' aA\', \'abBABba\'],
      dtype='|S7')
>>> (np.char.lstrip(c, \'\') == np.char.lstrip(c, None)).all()  # True
>>> (np.char.lstrip(c, \'\') == np.char.lstrip(c, None)).all()  # True
```
**partition** *(a, sep)*

Partition each element in `a` around `sep`. Calls `str.partition` element-wise.

For each element in `a`, split the element as the first occurrence of `sep`, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

**Parameters**
- `a`: array-like of str or unicode
- `sep`: str or unicode

**Returns**
- `out`: ndarray
  Output array of str or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

**See Also:**
- `str.partition`

**replace** *(a, old, new, count=None)*

For each element in `a`, return a copy of the string with all occurrences of substring `old` replaced by `new`. Calls `str.replace` element-wise.

**Parameters**
- `a`: array-like of str or unicode
- `old, new`: str or unicode
- `count`: int, optional
  If the optional argument `count` is given, only the first `count` occurrences are replaced.

**Returns**
- `out`: ndarray
  Output array of str or unicode, depending on input type

**See Also:**
- `str.replace`

**rjust** *(a, width, fillchar=' ')*

Return an array with the elements of `a` right-justified in a string of length `width`. Calls `str.rjust` element-wise.

**Parameters**
- `a`: array_like of str or unicode
- `width`: int
  The length of the resulting strings
- `fillchar`: str or unicode, optional
  The character to use for padding

**Returns**
- `out`: ndarray
  Output array of str or unicode, depending on input type
See Also:

str.rjust

rpartition(a, sep)
Partition each element in a around sep.
Calls str.partition element-wise.

For each element in a, split the element as the last occurrence of sep, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

Parameters
a : array-like of str or unicode
sep : str or unicode

Returns
out : ndarray
Output array of string or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

See Also:
str.rpartition

rsplit(a, sep=None, maxsplit=None)
For each element in a, return a list of the words in the string, using sep as the delimiter string.
Calls str.rsplit element-wise.

Except for splitting from the right, rsplit behaves like split.

Parameters
a : array_like of str or unicode
sep : str or unicode, optional
If sep is not specified or None, any whitespace string is a separator.
maxsplit : int, optional
If maxsplit is given, at most maxsplit splits are done, the rightmost ones.

Returns
out : ndarray
Array of list objects

See Also:
str.rsplit, split

rstrip(a, chars=None)
For each element in a, return a copy with the trailing characters removed.
Calls str.rstrip element-wise.

Parameters
a : array-like of str or unicode
chars : str or unicode, optional
The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped.

**Returns**
- `out`: ndarray

Output array of str or unicode, depending on input type

**See Also:**
- `str.rstrip`

**Examples**

```python
>>> c = np.array(['aAaAaA', 'abBABba'], dtype='S7'); c
array(['aAaAaA', 'abBABba'],
      dtype='|S7')
>>> np.char.rstrip(c, 'a')
array(['aAaAaA', 'abBABb'],
      dtype='|S6')
>>> np.char.rstrip(c, 'A')
array(['aAaAa', 'abBABba'],
      dtype='|S7')
```

**split** *(a, sep=None, maxsplit=None)*

For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.

Calls `str.rsplit` element-wise.

**Parameters**
- *a*: array_like of str or unicode
- *sep*: str or unicode, optional
  - If *sep* is not specified or None, any whitespace string is a separator.
- *maxsplit*: int, optional
  - If *maxsplit* is given, at most *maxsplit* splits are done.

**Returns**
- `out`: ndarray

Array of list objects

**See Also:**
- `str.split`, `rsplit`

**splitlines** *(a, keepends=None)*

For each element in *a*, return a list of the lines in the element, breaking at line boundaries.

Calls `str.splitlines` element-wise.

**Parameters**
- *a*: array_like of str or unicode
- *keepends*: bool, optional
  - Line breaks are not included in the resulting list unless *keepends* is given and true.

**Returns**
- `out`: ndarray
Array of list objects

See Also:

str.splitlines

strip(a, chars=None)
For each element in a, return a copy with the leading and trailing characters removed.
Calls str.rstrip element-wise.

Parameters

a : array-like of str or unicode
chars : str or unicode, optional

The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped.

Returns

out : ndarray
Output array of str or unicode, depending on input type

See Also:

str.strip

Examples

>>> c = np.array([‘aAaAa’, ‘ aA ’, ‘abBABba’])
>>> c
array([‘aAaAa’, ‘ aA’, ‘abBABba’],
dtype=’|S7’)
>>> np.char.strip(c)
array([‘aAaAa’, ‘aA’, ‘abBABba’],
dtype=’|S7’)
>>> np.char.strip(c, ‘a’) # ’a’ unstripped from c[1] because whitespace leads
array([‘aAaAa’, ‘aA’, ‘bBABb’],
dtype=’|S6’)
>>> np.char.strip(c, ‘A’) # ’A’ unstripped from c[1] because (unprinted) ws trails
array([‘aAaAa’, ‘aA’, ‘abBABba’],
dtype=’|S7’)

swapcase(a)
For each element in a, return a copy of the string with uppercase characters converted to lowercase and vice versa.
Calls str.swapcase element-wise.
For 8-bit strings, this method is locale-dependent.

Parameters

a : array-like of str or unicode

Returns

out : ndarray
Output array of str or unicode, depending on input type

See Also:

str.swapcase
Examples

```python
>>> c=np.array(['a1B c','1b Ca','b Ca1','cA1b'],'S5'); c
array(['a1B c', '1b Ca', 'b Ca1', 'cA1b'],
      dtype='|S5')
>>> np.char.swapcase(c)
array(['A1b C', '1B cA', 'B cA1', 'Ca1B'],
      dtype='|S5')
```

title(a)
For each element in a, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

Calls `str.title` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

- a : array-like of str or unicode

Returns

- out : ndarray

Output array of str or unicode, depending on input type

See Also:

- `str.title`

Examples

```python
>>> c=np.array(['a1b c','1b ca','b ca1','ca1b'],'S5'); c
array(['a1b c', '1b ca', 'b ca1', 'ca1b'],
      dtype='|S5')
>>> np.char.title(c)
chararray(['A1B C', '1B Ca', 'B Ca1', 'Ca1B'],
          dtype='|S5')
```

translate(a, table, deletechars=None)
For each element in a, return a copy of the string where all characters occurring in the optional argument `deletechars` are removed, and the remaining characters have been mapped through the given translation table.

Calls `str.translate` element-wise.

Parameters

- a : array-like of str or unicode
- table : str of length 256
- deletechars : str

Returns

- out : ndarray

Output array of str or unicode, depending on input type

See Also:

- `str.translate`

upper(a)
Return an array with the elements of a converted to uppercase.
Calls `str.upper` element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**
- `a` : array-like of str or unicode

**Returns**
- `out` : ndarray
  
  Output array of str or unicode, depending on input type

**See Also:**
- `str.upper`

**Examples**

```python
>>> c = np.array(['a1b c', '1bca', 'bca1']); c
array(['a1b c', '1bca', 'bca1'],
      dtype='|S5')
>>> numpy.char.upper(c)
array(['A1B C', '1BCA', 'BCA1'],
      dtype='|S5')
```

**zfill** (`a`, `width`)

Return the numeric string left-filled with zeros in a string of length `width`.

Calls `str.zfill` element-wise.

**Parameters**
- `a` : array-like of str or unicode
- `width` : int

**Returns**
- `out` : ndarray
  
  Output array of str or unicode, depending on input type

**See Also:**
- `str.zfill`

### 3.31.2 Comparison

Unlike the standard numpy comparison operators, the ones in the `char` module strip trailing whitespace characters before performing the comparison.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equal(x1, x2)</code></td>
<td>Return <code>(x1 == x2)</code> element-wise.</td>
</tr>
<tr>
<td><code>not_equal(x1, x2)</code></td>
<td>Return <code>(x1 != x2)</code> element-wise.</td>
</tr>
<tr>
<td><code>greater_equal(x1, x2)</code></td>
<td>Return <code>(x1 &gt;= x2)</code> element-wise.</td>
</tr>
<tr>
<td><code>less_equal(x1, x2)</code></td>
<td>Return <code>(x1 &lt;= x2)</code> element-wise.</td>
</tr>
<tr>
<td><code>greater(x1, x2)</code></td>
<td>Return <code>(x1 &gt; x2)</code> element-wise.</td>
</tr>
<tr>
<td><code>less(x1, x2)</code></td>
<td>Return <code>(x1 &lt; x2)</code> element-wise.</td>
</tr>
</tbody>
</table>

**equal** (`x1, x2`)

Return `(x1 == x2)` element-wise.

Unlike `numpy.equal`, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.
Parameters
   x1, x2 : array_like of str or unicode
           Input arrays of the same shape.

Returns
   out : {ndarray, bool}
           Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
   not_equal, greater_equal, less_equal, greater, less

not_equal(x1, x2)
Return (x1 != x2) element-wise.
Unlike numpy.not_equal, this comparison is performed by first stripping whitespace characters from
the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
   x1, x2 : array_like of str or unicode
           Input arrays of the same shape.

Returns
   out : {ndarray, bool}
           Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
   equal, greater_equal, less_equal, greater, less

greater_equal(x1, x2)
Return (x1 >= x2) element-wise.
Unlike numpy.greater_equal, this comparison is performed by first stripping whitespace characters from
the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
   x1, x2 : array_like of str or unicode
           Input arrays of the same shape.

Returns
   out : {ndarray, bool}
           Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
   equal, not_equal, less_equal, greater, less

less_equal(x1, x2)
Return (x1 <= x2) element-wise.
Unlike numpy.less_equal, this comparison is performed by first stripping whitespace characters from
the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
   x1, x2 : array_like of str or unicode
           Input arrays of the same shape.

Returns
   out : {ndarray, bool}
Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:

`equal`, `not_equal`, `greater_equal`, `greater`, `less`

greater(x1, x2)
Return (x1 > x2) element-wise.

Unlike `numpy.greater`, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode
    Input arrays of the same shape.

Returns

out : {ndarray, bool}
    Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:

`equal`, `not_equal`, `greater_equal`, `less_equal`, `less`

less(x1, x2)
Return (x1 < x2) element-wise.

Unlike `numpy.greater`, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters

x1, x2 : array_like of str or unicode
    Input arrays of the same shape.

Returns

out : {ndarray, bool}
    Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:

`equal`, `not_equal`, `greater_equal`, `less_equal`, `greater`

3.31. String operations
3.31.3 String information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count(a, sub[, start, end])</code></td>
<td>Returns an array with the number of non-overlapping occurrences of substring <code>sub</code> in the range <code>[start, end]</code>.</td>
</tr>
<tr>
<td><code>len</code></td>
<td>For each element, return the lowest index in the string where substring <code>sub</code> is found.</td>
</tr>
<tr>
<td><code>find(a, sub[, start, end])</code></td>
<td>Like <code>find</code>, but raises <code>ValueError</code> when the substring is not found.</td>
</tr>
<tr>
<td><code>index(a, sub[, start, end])</code></td>
<td>Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.</td>
</tr>
<tr>
<td><code>isalpha(a)</code></td>
<td>Returns true for each element in <code>a</code> that are only decimal characters in the element.</td>
</tr>
<tr>
<td><code>isdigit(a)</code></td>
<td>Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.</td>
</tr>
<tr>
<td><code>islower(a)</code></td>
<td>Returns true for each element if all cased characters in the string are lowercase and there is at least one character, false otherwise.</td>
</tr>
<tr>
<td><code>isnumeric(a)</code></td>
<td>Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.</td>
</tr>
<tr>
<td><code>isspace(a)</code></td>
<td>Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.</td>
</tr>
<tr>
<td><code>istitle(a)</code></td>
<td>Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.</td>
</tr>
<tr>
<td><code>rfind(a, sub[, start, end])</code></td>
<td>For each element in <code>a</code>, return the highest index in the string where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>[start, end]</code>.</td>
</tr>
<tr>
<td><code>rindex(a, sub[, start, end])</code></td>
<td>Like <code>rfind</code>, but raises <code>ValueError</code> when the substring <code>sub</code> is not found.</td>
</tr>
<tr>
<td><code>startswith(a, prefix[, start, end])</code></td>
<td>Returns a boolean array which is <code>True</code> where the string element in <code>a</code> starts with <code>prefix</code>, otherwise <code>False</code>.</td>
</tr>
</tbody>
</table>

`count` ([`a`, `sub`, `start=0`, `end=None`])

Returns an array with the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Calls `str.count` element-wise.

Parameters

- `a`: array_like of str or unicode
- `sub`: str or unicode
  The substring to search for.
- `start`, `end`: int, optional
  Optional arguments `start` and `end` are interpreted as slice notation to specify the range in which to count.

Returns

- `out`: ndarray
  Output array of ints.

See Also:

- `str.count`

Examples

```python
>>> c = np.array(['aAaAaA', ' aA ', 'abBAbba'])
>>> c
array(['aAaAaA', ' aA ', 'abBAbba'],
      dtype='|S7')
```
>>> np.char.count(c, 'A')
array([3, 1, 1])
>>> np.char.count(c, 'aA')
array([3, 1, 0])
>>> np.char.count(c, 'A', start=1, end=4)
array([2, 1, 1])
>>> np.char.count(c, 'A', start=1, end=3)
array([1, 0, 0])

**find** *(a, sub, start=0, end=None)*

For each element, return the lowest index in the string where substring *sub* is found.

Calls `str.find` element-wise.

For each element, return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range `[start, end]`.

**Parameters**

- **a**: array_like of str or unicode
- **sub**: str or unicode
- **start, end**: int, optional
  
  Optional arguments *start* and *end* are interpreted as in slice notation.

**Returns**

- **out**: ndarray or int
  
  Output array of ints. Returns -1 if *sub* is not found.

**See Also:**

`str.find`

**index** *(a, sub, start=0, end=None)*

Like `find`, but raises `ValueError` when the substring is not found.

Calls `str.index` element-wise.

**Parameters**

- **a**: array_like of str or unicode
- **sub**: str or unicode
- **start, end**: int, optional

**Returns**

- **out**: ndarray
  
  Output array of ints. Returns -1 if *sub* is not found.

**See Also:**

`find`, `str.find`

**isalpha** *(a)*

Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

Calls `str.isalpha` element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

- **a**: array_like of str or unicode

3.31. String operations

827
isdecimal \((a)\)

For each element in \(a\), return True if there are only decimal characters in the element.

Calls `unicode.isdecimal` element-wise.

Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

**Parameters**
- \(a\) : array-like of unicode

**Returns**
- \(out\) : ndarray
  
  Array of bools

See Also:
- `unicode.isdecimal`

isdigit \((a)\)

Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

Calls `str.isdigit` element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**
- \(a\) : array_like of str or unicode

**Returns**
- \(out\) : ndarray
  
  Output array of bools

See Also:
- `str.isdigit`

islower \((a)\)

Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Calls `str.islower` element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**
- \(a\) : array_like of str or unicode

**Returns**
- \(out\) : ndarray
  
  Output array of bools
See Also:

`str.islower`

`isnumeric(a)`
For each element in `a`, return True if there are only numeric characters in the element.
Calls `unicode.isnumeric` element-wise.

Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH.

Parameters

- `a` : array-like of unicode

Returns

- `out` : ndarray

Array of booleans

See Also:

`unicode.isnumeric`

`isspace(a)`
Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.
Calls `str.isspace` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

- `a` : array_like of str or unicode

Returns

- `out` : ndarray

Output array of bools

See Also:

`str.isspace`

`istitle(a)`
Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.
Call `str.istitle` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

- `a` : array_like of str or unicode

Returns

- `out` : ndarray

Output array of bools

See Also:

`str.istitle`

`isupper(a)`
Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.
Call `str.isupper` element-wise.
For 8-bit strings, this method is locale-dependent.

**Parameters**
- `a` : array_like of str or unicode

**Returns**
- `out` : ndarray
  Output array of bools

**See Also:**
- `str.isupper`

### `rfind(a, sub, start=0, end=None)`
For each element in `a`, return the highest index in the string where substring `sub` is found, such that `sub` is contained within `[start, end]`.
Calls `str.rfind` element-wise.

**Parameters**
- `a` : array-like of str or unicode
- `sub` : str or unicode
- `start, end` : int, optional
  Optional arguments `start` and `end` are interpreted as in slice notation.

**Returns**
- `out` : ndarray
  Output array of ints. Return -1 on failure.

**See Also:**
- `str.rfind`

### `rindex(a, sub, start=0, end=None)`
Like `rfind`, but raises `ValueError` when the substring `sub` is not found.
Calls `str.rindex` element-wise.

**Parameters**
- `a` : array-like of str or unicode
- `sub` : str or unicode
- `start, end` : int, optional

**Returns**
- `out` : ndarray
  Output array of ints.

**See Also:**
- `rfind`, `str.rindex`

### `startswith(a, prefix, start=0, end=None)`
Returns a boolean array which is `True` where the string element in `a` starts with `prefix`, otherwise `False`.
Calls `str.startswith` element-wise.
Parameters
   a : array_like of str or unicode
   suffix : str
   start, end : int, optional
      With optional start, test beginning at that position. With optional end, stop comparing
      at that position.

Returns
   out : ndarray
      Array of booleans

See Also:
   str.startswith

3.31.4 Convenience class

| chararray (shape[, itemsize, unicode, ...]) | Provides a convenient view on arrays of string and unicode values. |
| class chararray () | Provides a convenient view on arrays of string and unicode values. |

Note: The chararray class exists for backwards compatibility with Numarray, it is not recommended for new
development. If one needs arrays of strings, use arrays of dtype object_, string_ or unicode_, and use the free
functions in the numpy.char module for fast vectorized string operations.

Versus a regular Numpy array of type str or unicode, this class adds the following functionality:
   1.values automatically have whitespace removed from the end when indexed
   2.comparison operators automatically remove whitespace from the end when comparing values
   3.vectorized string operations are provided as methods (e.g. endswith) and infix operators (e.g. "+", "*", 
      ")

cchararrays should be created using numpy.char.array or numpy.char.asarray, rather than this con-
structor directly.

This constructor creates the array, using buffer (with offset and strides) if it is not None. If buffer is None, then
constructs a new array with strides in “C order”, unless both len(shape) >= 2 and order=’Fortran’,
in which case strides is in “Fortran order”.

Parameters
   shape : tuple
      Shape of the array.
   itemsize : int, optional
      Length of each array element, in number of characters. Default is 1.
   unicode : bool, optional
      Are the array elements of type unicode (True) or string (False). Default is False.
   buffer : int, optional
      Memory address of the start of the array data. Default is None, in which case a new
      array is created.
   offset : int, optional

3.31. String operations
Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be >=0.

**strides**: array_like of ints, optional

Strides for the array (see `ndarray.strides` for full description). Default is None.

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

The order in which the array data is stored in memory: ‘C’ -> “row major” order (the default), ‘F’ -> “column major” (Fortran) order.

### Examples

```python
cchar = np.chararray((3, 3))
cchar[:, :] = 'abc'
cchar
```

```
chararray([[‘a’, ‘a’, ‘a’],
         [‘a’, ‘a’, ‘a’],
         [‘a’, ‘a’, ‘a’]],
dtype=‘|S1’)
```

```python
cchar = np.chararray(cchar.shape, itemsize=5)
cchar[:, :] = 'abc'
cchar
```

```
chararray([[‘abc’, ‘abc’, ‘abc’],
         [‘abc’, ‘abc’, ‘abc’],
         [‘abc’, ‘abc’, ‘abc’]],
dtype=‘|S5’)
```

### Methods
PACKAGING (NUMPY.DISTUTILS)

NumPy provides enhanced distutils functionality to make it easier to build and install sub-packages, auto-generate code, and extension modules that use Fortran-compiled libraries. To use features of NumPy distutils, use the setup command from numpy.distutils.core. A useful Configuration class is also provided in numpy.distutils.misc_util that can make it easier to construct keyword arguments to pass to the setup function (by passing the dictionary obtained from the todict() method of the class). More information is available in the NumPy Distutils Users Guide in <site-packages>/numpy/doc/DISTUTILS.txt.

4.1 Modules in numpy.distutils

4.1.1 misc_util

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_numpy_include_dirs()</td>
<td>Convert a /-separated pathname to one using the OS’s path separator.</td>
</tr>
<tr>
<td>dict_append(d, **kws)</td>
<td></td>
</tr>
<tr>
<td>appendpath(prefix, path)</td>
<td></td>
</tr>
<tr>
<td>allpath(name)</td>
<td></td>
</tr>
<tr>
<td>dot_joined(*args)</td>
<td></td>
</tr>
<tr>
<td>generate_config_py (tar)</td>
<td>Generate config.py file containing system_info information used during building the package.</td>
</tr>
<tr>
<td>get_cmd(cmd-name[, _cache])</td>
<td></td>
</tr>
<tr>
<td>terminal_has_colors()</td>
<td></td>
</tr>
<tr>
<td>red_text(s)</td>
<td></td>
</tr>
<tr>
<td>green_text(s)</td>
<td></td>
</tr>
<tr>
<td>yellow_text(s)</td>
<td></td>
</tr>
<tr>
<td>blue_text(s)</td>
<td></td>
</tr>
<tr>
<td>cyan_text(s)</td>
<td></td>
</tr>
<tr>
<td>cygwin32(path)</td>
<td></td>
</tr>
<tr>
<td>all_strings(lst)</td>
<td>Return True if all items in lst are string objects.</td>
</tr>
<tr>
<td>has_f_sources(sources)</td>
<td>Return True if sources contains Fortran files</td>
</tr>
<tr>
<td>has_cxx_sources(sources)</td>
<td>Return True if sources contains C++ files</td>
</tr>
<tr>
<td>filter_sources(sources)</td>
<td>Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.</td>
</tr>
<tr>
<td>get_dependencies(sources)</td>
<td></td>
</tr>
<tr>
<td>is_local_src_dir(directory)</td>
<td>Return true if directory is local directory.</td>
</tr>
<tr>
<td>get_ext_source_files(ext)</td>
<td></td>
</tr>
<tr>
<td>get_script_files(scripts)</td>
<td></td>
</tr>
</tbody>
</table>
class Configuration(
    package_name=None, parent_name=None, top_path=None, package_path=None,
    caller_level=1, setup_name='setup.py', **attrs)

def get_numpy_include_dirs()

def dict_append(d, **kws)

def appendpath(prefix, path)

def allpath(name)
    Convert a /-separated pathname to one using the OS's path separator.

def dot_join(*args)

def generate_config_py(target)
    Generate config.py file containing system_info information used during building the package.

    Usage:
    config['py_modules'].append((packagename, '__config__', generate_config_py))

def get_cmd(cmdname, _cache={})

def terminal_has_colors()

def red_text(s)

def green_text(s)

def yellow_text(s)

def blue_text(s)

def cyan_text(s)

def cyg2win32(path)

def all_strings(lst)
    Return True if all items in lst are string objects.

def has_f_sources(sources)
    Return True if sources contains Fortran files

def has_cxx_sources(sources)
    Return True if sources contains C++ files

def filter_sources(sources)
    Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.

def get_dependencies(sources)
**is_local_src_dir** *(directory)*
Return true if directory is local directory.

**get_ext_source_files** *(ext)*

**get_script_files** *(scripts)*

**class Configuration** *(package_name=None, parent_name=None, top_path=None, package_path=None, **attrs)*
Construct a configuration instance for the given package name. If `parent_name` is not `None`, then construct the package as a sub-package of the `parent_name` package. If `top_path` and `package_path` are `None` then they are assumed equal to the path of the file this instance was created in. The setup.py files in the numpy distribution are good examples of how to use the `Configuration` instance.

**todict** ()
Return a dictionary compatible with the keyword arguments of distutils setup function.

**Examples**

```python
>>> setup(**config.todict()).
```

**get_distribution** ()
Return the distutils distribution object for self.

**get_subpackage** *(subpackage_name, subpackage_path=None, parent_name=None, caller_level=1)*
Return list of subpackage configurations.

**Parameters**

- `subpackage_name`: str, `None`
  Name of the subpackage to get the configuration. ‘*’ in subpackage_name is handled as a wildcard.

- `subpackage_path`: str
  If `None`, then the path is assumed to be the local path plus the subpackage_name. If a setup.py file is not found in the subpackage_path, then a default configuration is used.

- `parent_name`: str
  Parent name.

**add_subpackage** *(subpackage_name, subpackage_path=None, standalone=False)*
Add a sub-package to the current Configuration instance.

This is useful in a setup.py script for adding sub-packages to a package.

**Parameters**

- `subpackage_name`: str
  name of the subpackage

- `subpackage_path`: str
  if given, the subpackage path such as the subpackage is in subpackage_path / subpackage_name. If None, the subpackage is assumed to be located in the local path / subpackage_name.

- `standalone`: bool

**add_data_files** *(files)*
Add data files to configuration data_files.
Parameters

files: sequence :

Argument(s) can be either

• 2-sequence (<datadir prefix>,<path to data file(s)>)

• paths to data files where python datadir prefix defaults to package dir.

Notes

The form of each element of the files sequence is very flexible allowing many combinations of where to get the files from the package and where they should ultimately be installed on the system. The most basic usage is for an element of the files argument sequence to be a simple filename. This will cause that file from the local path to be installed to the installation path of the self.name package (package path). The file argument can also be a relative path in which case the entire relative path will be installed into the package directory. Finally, the file can be an absolute path name in which case the file will be found at the absolute path name but installed to the package path.

This basic behavior can be augmented by passing a 2-tuple in as the file argument. The first element of the tuple should specify the relative path (under the package install directory) where the remaining sequence of files should be installed to (it has nothing to do with the file-names in the source distribution). The second element of the tuple is the sequence of files that should be installed. The files in this sequence can be filenames, relative paths, or absolute paths. For absolute paths the file will be installed in the top-level package installation directory (regardless of the first argument). Filenames and relative path names will be installed in the package install directory under the path name given as the first element of the tuple.

Rules for installation paths:

1. file.txt -> (. , file.txt)-> parent/file.txt
2. foo/file.txt -> (foo, foo/file.txt) -> parent/foo/file.txt
3. ./foo/bar/file.txt -> (. , ./foo/bar/file.txt) -> parent/file.txt
4. *.txt -> parent/a.txt, parent/b.txt
5. foo/*.txt -> parent/foo/a.txt, parent/foo/b.txt
6. ./txt -> (. , */txt) -> parent/c/a.txt, parent/d/b.txt
7. (sun, file.txt) -> parent/sun/file.txt
8. (sun, bar/file.txt) -> parent/sun/file.txt
9. (sun, /foo/bar/file.txt) -> parent/sun/file.txt
10. (sun, */*.txt) -> parent/sun/a.txt, parent/sun/b.txt
11. (sun, bar/*.txt) -> parent/sun/a.txt, parent/sun/b.txt
12. (sun/ , */*.txt) -> parent/sun/c/a.txt, parent/d/b.txt

An additional feature is that the path to a data-file can actually be a function that takes no arguments and returns the actual path(s) to the data-files. This is useful when the data files are generated while building the package.

Examples

Add files to the list of data_files to be included with the package.

```python
>>> self.add_data_files('foo.dat',
                       ('fun', ['gun.dat', 'nun/pun.dat', '/tmp/sun.dat']),
```

Chapter 4. Packaging (numpy.distutils)
will install these data files to:

<package install directory>/
foo.dat
fun/
  gun.dat
  nun/
    pun.dat
sun.dat
bar/
  car.dat
  can.dat

where <package install directory> is the package (or sub-package) directory such as
'/usr/lib/python2.4/site-packages/mypackage' ('C: Python2.4 Lib site-packages mypackage') or
'/usr/lib/python2.4/site-packages/mypackage/mysubpackage' ('C: Python2.4 Lib site-packages my-
package mysubpackage').

add_data_dir(data_path)
Recursively add files under data_path to data_files list.

Recursively add files under data_path to the list of data_files to be installed (and distributed). The data_path
can be either a relative path-name, or an absolute path-name, or a 2-tuple where the first argument shows
where in the install directory the data directory should be installed to.

Parameters
data_path: seq,str :
  Argument can be either
  • 2-sequence (<datadir suffix>,<path to data directory>)
  • path to data directory where python datadir suffix defaults to package dir.

Notes
Rules for installation paths:
  foo/bar -> (foo/bar, foo/bar) -> parent/foo/bar (gun, foo/bar) -> parent/gun foo/* -> (foo/a, foo/a),
  (foo/b, foo/b) -> parent/foo/a, parent/foo/b (gun, foo/) -> (gun, foo/a), (gun, foo/b) -> gun (gun/, foo/)
  -> parent/gun/a, parent/gun/b /foo/bar -> (bar, /foo/bar) -> parent/bar (gun, /foo/bar) -> parent/gun
  (fun/*, sun/ /foo/bar) -> parent/fun/foo/gun/bar

Examples
For example suppose the source directory contains fun/foo.dat and fun/bar/car.dat:

```python
>>> self.add_data_dir('fun')
>>> self.add_data_dir(('sun', 'fun'))
>>> self.add_data_dir(('gun', '/full/path/to/fun'))
```

Will install data-files to the locations:

<package install directory>/
fun/
  foo.dat
bar/
add_include_dirs(*paths)
   Add paths to configuration include directories.
   Add the given sequence of paths to the beginning of the include_dirs list. This list will be visible to all
   extension modules of the current package.

add_headers(*files)
   Add installable headers to configuration.
   Add the given sequence of files to the beginning of the headers list. By default, headers will be installed
   under <python-include> retain name.replace(‘.’,’’)/ directory. If an item of files is a tuple, then its first
   argument specifies the actual installation location relative to the <python-include> path.

Parameters
   files: str, seq :
      Argument(s) can be either:
      • 2-sequence (<includedir suffix>,<path to header file(s)>)
      • path(s) to header file(s) where python includedir suffix will default to package name.

add_extension(name, sources, **kw)
   Add extension to configuration.
   Create and add an Extension instance to the ext_modules list. This method also takes the following optional
   keyword arguments that are passed on to the Extension constructor.

Parameters
   name: str :
      name of the extension
   sources: seq :
      list of the sources. The list of sources may contain functions (called source generators)
      which must take an extension instance and a build directory as inputs and return a source
      file or list of source files or None. If None is returned then no sources are generated.
      If the Extension instance has no sources after processing all source generators, then no
      extension module is built.

   include_dirs: :
   define_macros: :
   undef_macros: :
   library_dirs: :
   libraries: :
   runtime_library_dirs: :
   extra_objects: :
   extra_compile_args: :
extra_link_args: :
export_symbols: :
swig_opts: :
depends: :
The depends list contains paths to files or directories that the sources of the extension module depend on. If any path in the depends list is newer than the extension module, then the module will be rebuilt.
language: :
f2py_options: :
module_dirs: :
extra_info: dict,list:
dict or list of dict of keywords to be appended to keywords.

Notes
The self.paths(...) method is applied to all lists that may contain paths.

add_library (name, sources, **build_info)
Add library to configuration.

Parameters
name : str
Name of the extension.
sources : sequence
List of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.

build_info : dict, optional
The following keys are allowed:
• depends
• macros
• include_dirs
• extra_compiler_args
• f2py_options
• language

add_scripts (*files)
Add scripts to configuration.

Add the sequence of files to the beginning of the scripts list. Scripts will be installed under the <prefix>/bin/directory.

add_installed_library (name, sources, install_dir, build_info=None)
Similar to add_library, but the specified library is installed.

4.1. Modules in numpy.distutils
Most C libraries used with distutils are only used to build python extensions, but libraries built through this method will be installed so that they can be reused by third-party packages.

**Parameters**

- **name** : str
  Name of the installed library.

- **sources** : sequence
  List of the library’s source files. See `add_library` for details.

- **install_dir** : str
  Path to install the library, relative to the current sub-package.

- **build_info** : dict, optional
  The following keys are allowed:
  - `depends`
  - `macros`
  - `include_dirs`
  - `extra_compiler_args`
  - `f2py_options`
  - `language`

**Returns**

- None

**See Also:**

- `add_library`, `add_npy_pkg_config`, `get_info`

**Notes**

The best way to encode the options required to link against the specified C libraries is to use a “libname.ini” file, and use `get_info` to retrieve the required options (see `add_npy_pkg_config` for more information).

**add_npy_pkg_config**(template, install_dir, subst_dict=None)

Generate and install a npy-pkg config file from a template.

The config file generated from `template` is installed in the given install directory, using `subst_dict` for variable substitution.

**Parameters**

- **template** : str
  The path of the template, relatively to the current package path.

- **install_dir** : str
  Where to install the npy-pkg config file, relatively to the current package path.

- **subst_dict** : dict, optional
  If given, any string of the form `@key@` will be replaced by `subst_dict[key]` in the template file when installed. The install prefix is always available through the variable `@prefix@`, since the install prefix is not easy to get reliably from setup.py.

**See Also:**

- `add_installed_library`, `get_info`
Notes

This works for both standard installs and in-place builds, i.e. the `@prefix@` refer to the source directory for in-place builds.

Examples

```python
config.add_npy_pkg_config('foo.ini.in', 'lib', {'foo': bar})
```

Assuming the `foo.ini.in` file has the following content:

```ini
[meta]
Name=@foo@
Version=1.0
Description=dummy description

[default]
Cflags=-I@prefix@/include
Libs=
```

The generated file will have the following content:

```ini
[meta]
Name=bar
Version=1.0
Description=dummy description

[default]
Cflags=-Iprefix_dir/include
Libs=
```

and will be installed as foo.ini in the ‘lib’ subpath.

paths(*paths, **kws)

Apply glob to paths and prepend local_path if needed.

Applies `glob.glob(...)` to each path in the sequence (if needed) and pre-pends the local_path if needed. Because this is called on all source lists, this allows wildcard characters to be specified in lists of sources for extension modules and libraries and scripts and allows path-names be relative to the source directory.

get_config_cmd()

Returns the numpy.distutils config command instance.

get_build_temp_dir()

Return a path to a temporary directory where temporary files should be placed.

have_f77c()

Check for availability of Fortran 77 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.

Notes

True if a Fortran 77 compiler is available (because a simple Fortran 77 code was able to be compiled successfully).

have_f90c()

Check for availability of Fortran 90 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.
Notes

True if a Fortran 90 compiler is available (because a simple Fortran 90 code was able to be compiled successfully)

get_version (version_file=None, version_variable=None)
Try to get version string of a package.

Return a version string of the current package or None if the version information could not be detected.

Notes

This method scans files named __version__.py, <packagename>_version.py, version.py, and __svn_version__.py for string variables version, __version__, and <packagename>_version, until a version number is found.

make_svn_version_py (delete=True)
Appends a data function to the data_files list that will generate __svn_version__.py file to the current package directory.

Generate package __svn_version__.py file from SVN revision number, it will be removed after python exits but will be available when sdist, etc commands are executed.

Notes

If __svn_version__.py existed before, nothing is done.

This is intended for working with source directories that are in an SVN repository.

make_config_py (name='__config__')
Generate package __config__.py file containing system_info information used during building the package.

This file is installed to the package installation directory.

get_info (*names)
Get resources information.

Return information (from system_info.get_info) for all of the names in the argument list in a single dictionary.

4.1.2 Other modules

system_info.get_info(, notfound_action=0)

notfound_action: 0 - do nothing 1 - display warning message 2 - raise error

system_info.get_standard_file(fname)
Returns a list of files named ‘fname’ from 1) System-wide directory (directory-location of this module) 2) Users HOME directory (os.environ[‘HOME’]) 3) Local directory

cpuinfo.cpu

log.set_verbosity(v[, force])

exec_command

get_info (name, notfound_action=0)

notfound_action:

0 - do nothing 1 - display warning message 2 - raise error

get_standard_file (fname)

Returns a list of files named ‘fname’ from 1) System-wide directory (directory-location of this module) 2) Users HOME directory (os.environ[‘HOME’]) 3) Local directory
4.2 Building Installable C libraries

Conventional C libraries (installed through `add_library`) are not installed, and are just used during the build (they are statically linked). An installable C library is a pure C library, which does not depend on the python C runtime, and is installed such that it may be used by third-party packages. To build and install the C library, you just use the method `add_installed_library` instead of `add_library`, which takes the same arguments except for an additional `install_dir` argument:

```python
>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
```

4.2.1 npy-pkg-config files

To make the necessary build options available to third parties, you could use the `npy-pkg-config` mechanism implemented in `numpy.distutils`. This mechanism is based on a .ini file which contains all the options. A .ini file is very similar to .pc files as used by the pkg-config unix utility:

```
[meta]
Name: foo
Version: 1.0
Description: foo library

[variables]
prefix = /home/user/local
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

Generally, the file needs to be generated during the build, since it needs some information known at build time only (e.g. prefix). This is mostly automatic if one uses the `Configuration` method `add_npy_pkg_config`. Assuming we have a template file foo.ini.in as follows:

```
[meta]
Name: foo
Version: @version@
Description: foo library

[variables]
prefix = @prefix@
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

4.2. Building Installable C libraries
and the following code in setup.py:

```python
cfg.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
cfg.subst = {'version': '1.0'}
cfg.add_npy_pkg_config('foo.ini.in', 'lib', subst_dict=subst)
```

This will install the file foo.ini into the directory package_dir/lib, and the foo.ini file will be generated from foo.ini.in, where each `@version@` will be replaced by `subst_dict['version']`. The dictionary has an additional prefix substitution rule automatically added, which contains the install prefix (since this is not easy to get from setup.py). 

**4.2.2 Reusing a C library from another package**

Info are easily retrieved from the `get_info` function in `numpy.distutils.misc_util`:

```python
info = get_info('npymath')
cfg.add_extension('foo', sources=['foo.c'], extra_info=info)
```

An additional list of paths to look for .ini files can be given to `get_info`.

**4.3 Conversion of .src files**

NumPy distutils supports automatic conversion of source files named `<somefile>.src`. This facility can be used to maintain very similar code blocks requiring only simple changes between blocks. During the build phase of setup, if a template file named `<somefile>.src` is encountered, a new file named `<somefile>` is constructed from the template and placed in the build directory to be used instead. Two forms of template conversion are supported. The first form occurs for files named named `<file>.ext.src` where ext is a recognized Fortran extension (f, f90, f95, f77, for, ftn, pyf). The second form is used for all other cases.

**4.3.1 Fortran files**

This template converter will replicate all `function` and `subroutine` blocks in the file with names that contain `<...>` according to the rules in `<...>`: The number of comma-separated words in `<...>` determines the number of times the block is repeated. What these words are indicates what that repeat rule, `<...>`, should be replaced with in each block. All of the repeat rules in a block must contain the same number of comma-separated words indicating the number of times that block should be repeated. If the word in the repeat rule needs a comma, leftarrow, or rightarrow, then prepend it with a backslash `\`. If a word in the repeat rule matches `\<index>` then it will be replaced with the `<index>`-th word in the same repeat specification. There are two forms for the repeat rule: named and short.

**Named repeat rule**

A named repeat rule is useful when the same set of repeats must be used several times in a block. It is specified using `<rule1=item1, item2, item3,..., itemN>`, where N is the number of times the block should be repeated. On each repeat of the block, the entire expression, `<...>` will be replaced first with item1, and then with item2, and so forth until N repeats are accomplished. Once a named repeat specification has been introduced, the same repeat rule may be used in the current block by referring only to the name (i.e. `<rule1>`).
Short repeat rule

A short repeat rule looks like <item1, item2, item3, ..., itemN>. The rule specifies that the entire expression, ‘<...>’ should be replaced first with item1, and then with item2, and so forth until N repeats are accomplished.

Pre-defined names

The following predefined named repeat rules are available:

- <prefix=s,d,c,z>
- <c=s,d,c,z>
- <_t=real, double precision, complex, double complex>
- <ftype=real, double precision, complex, double complex>
- <ctype=float, double, complex_float, complex_double>
- <ftypereal=float, double precision, \0, \1>
- <ctypereal=float, double, \0, \1>

4.3.2 Other files

Non-Fortran files use a separate syntax for defining template blocks that should be repeated using a variable expansion similar to the named repeat rules of the Fortran-specific repeats. The template rules for these files are:

1. “/**begin repeat “on a line by itself marks the beginning of a segment that should be repeated.

2. Named variable expansions are defined using #name=item1, item2, item3, ..., itemN# and placed on successive lines. These variables are replaced in each repeat block with corresponding word. All named variables in the same repeat block must define the same number of words.

3. In specifying the repeat rule for a named variable, item*N is short-hand for item, item, ..., item repeated N times. In addition, parenthesis in combination with *N can be used for grouping several items that should be repeated. Thus, #name=(item1, item2)*4# is equivalent to #name=item1, item2, item1, item2, item1, item2, item1, item2#

4. “*/ “on a line by itself marks the end of the the variable expansion naming. The next line is the first line that will be repeated using the named rules.

5. Inside the block to be repeated, the variables that should be expanded are specified as @name@.

6. “/**end repeat**/ “on a line by itself marks the previous line as the last line of the block to be repeated.
NumPy provides a C-API to enable users to extend the system and get access to the array object for use in other routines. The best way to truly understand the C-API is to read the source code. If you are unfamiliar with (C) source code, however, this can be a daunting experience at first. Be assured that the task becomes easier with practice, and you may be surprised at how simple the C-code can be to understand. Even if you don’t think you can write C-code from scratch, it is much easier to understand and modify already-written source code than create it de novo.

Python extensions are especially straightforward to understand because they all have a very similar structure. Admittedly, NumPy is not a trivial extension to Python, and may take a little more snooping to grasp. This is especially true because of the code-generation techniques, which simplify maintenance of very similar code, but can make the code a little less readable to beginners. Still, with a little persistence, the code can be opened to your understanding. It is my hope, that this guide to the C-API can assist in the process of becoming familiar with the compiled-level work that can be done with NumPy in order to squeeze that last bit of necessary speed out of your code.

5.1 Python Types and C-Structures

Several new types are defined in the C-code. Most of these are accessible from Python, but a few are not exposed due to their limited use. Every new Python type has an associated PyObject * with an internal structure that includes a pointer to a “method table” that defines how the new object behaves in Python. When you receive a Python object into C code, you always get a pointer to a PyObject structure. Because a PyObject structure is very generic and defines only PyObject_HEAD, by itself it is not very interesting. However, different objects contain more details after the PyObject_HEAD (but you have to cast to the correct type to access them — or use accessor functions or macros).

5.1.1 New Python Types Defined

Python types are the functional equivalent in C of classes in Python. By constructing a new Python type you make available a new object for Python. The ndarray object is an example of a new type defined in C. New types are defined in C by two basic steps:

1. creating a C-structure (usually named Py{Name}Object) that is binary- compatible with the PyObject structure itself but holds the additional information needed for that particular object;

2. populating the PyTypeObject table (pointed to by the ob_type member of the PyObject structure) with pointers to functions that implement the desired behavior for the type.
Instead of special method names which define behavior for Python classes, there are “function tables” which point to
functions that implement the desired results. Since Python 2.2, the PyTypeObject itself has become dynamic which
allows C types that can be “sub-typed “from other C-types in C, and sub-classed in Python. The children types inherit
the attributes and methods from their parent(s).

There are two major new types: the ndarray (PyArray_Type) and the ufunc (PyUFunc_Type). Ad-
ditional types play a supportive role: the PyArrayIter_Type, the PyArrayMultiIter_Type, and the
PyArrayDescr_Type. The PyArrayIter_Type is the type for a flat iterator for an ndarray (the object that
is returned when getting the flat attribute). The PyArrayMultiIter_Type is the type of the object returned
when calling broadcast(). It handles iteration and broadcasting over a collection of nested sequences. Also, the
PyArrayDescr_Type is the data-type-descriptor type whose instances describe the data. Finally, there are 21 new
scalar-array types which are new Python scalars corresponding to each of the fundamental data types available for
arrays. An additional 10 other types are place holders that allow the array scalars to fit into a hierarchy of actual
Python types.

PyArray_Type

The Python type of the nd is PyArray_Type. In C, every ndarray is a pointer to a PyArrayObject
structure. The ob_type member of this structure contains a pointer to the PyArray_Type typeobject.

PyArrayObject

The PyArrayObject C-structure contains all of the required information for an array. All instances of an
ndarray (and its subclasses) will have this structure. For future compatibility, these structure members should
normally be accessed using the provided macros. If you need a shorter name, then you can make use of
NPY_AO which is defined to be equivalent to PyArrayObject.

typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject;

PyObject_HEAD
This is needed by all Python objects. It consists of (at least) a reference count member (ob_refcnt) and
a pointer to the typeobject (ob_type). (Other elements may also be present if Python was compiled with
special options see Include/object.h in the Python source tree for more information). The ob_type member
points to a Python type object.

char data
A pointer to the first element of the array. This pointer can (and normally should) be recast to the data type of
the array.

int nd
An integer providing the number of dimensions for this array. When nd is 0, the array is sometimes called a
rank-0 array. Such arrays have undefined dimensions and strides and cannot be accessed. NPY_MAXDIMS is
the largest number of dimensions for any array.

npy_intp dimensions
An array of integers providing the shape in each dimension as long as nd ≥ 1. The integer is always large
enough to hold a pointer on the platform, so the dimension size is only limited by memory.
**npy_intp strides**

An array of integers providing for each dimension the number of bytes that must be skipped to get to the next element in that dimension.

**PyObject base**

This member is used to hold a pointer to another Python object that is related to this array. There are two use cases: 1) If this array does not own its own memory, then base points to the Python object that owns it (perhaps another array object), 2) If this array has the `NPY_UPDATEIFCOPY` flag set, then this array is a working copy of a “misbehaved” array. As soon as this array is deleted, the array pointed to by base will be updated with the contents of this array.

**PyArray_Descr descr**

A pointer to a data-type descriptor object (see below). The data-type descriptor object is an instance of a new built-in type which allows a generic description of memory. There is a descriptor structure for each data type supported. This descriptor structure contains useful information about the type as well as a pointer to a table of function pointers to implement specific functionality.

**int flags**

Flags indicating how the memory pointed to by data is to be interpreted. Possible flags are `NPY_C_CONTIGUOUS`, `NPY_F_CONTIGUOUS`, `NPY_OWNDATA`, `NPY_ALIGNED`, `NPY_WRITEABLE`, and `NPY_UPDATEIFCOPY`.

**PyObject weakreflist**

This member allows array objects to have weak references (using the weakref module).

**PyArrayDescr_Type**

The `PyArrayDescr_Type` is the built-in type of the data-type-descriptor objects used to describe how the bytes comprising the array are to be interpreted. There are 21 statically-defined `PyArray_Descr` objects for the built-in data-types. While these participate in reference counting, their reference count should never reach zero. There is also a dynamic table of user-defined `PyArray_Descr` objects that is also maintained. Once a data-type-descriptor object is “registered” it should never be deallocated either. The function `PyArray_DescrFromType(...)` can be used to retrieve a `PyArray_Descr` object from an enumerated type-number (either built-in or user-defined).

**PyArray_Descr**

The format of the `PyArray_Descr` structure that lies at the heart of the `PyArrayDescr_Type` is

```c
typedef struct {
    PyObject_HEAD
    PyTypeObject *typeobj;
    char kind;
    char type;
    char byteorder;
    char hasobject;
    int type_num;
    int elsize;
    int alignment;
    PyArray_ArrayDescr *subarray;
    PyObject *fields;
    PyArray_ArrFuncs *f;
} PyArray_Descr;
```

**PyTypeObject typeobj**

Pointer to a typeobject that is the corresponding Python type for the elements of this array. For the built-in types, this points to the corresponding array scalar. For user-defined types, this should point to a user-defined
typeobject. This typeobject can either inherit from array scalars or not. If it does not inherit from array scalars, then the `NPY_USE_GETITEM` and `NPY_USE_SETITEM` flags should be set in the `hasobject` flag.

`char kind`
A character code indicating the kind of array (using the array interface typestring notation). A ‘b’ represents Boolean, a ‘i’ represents signed integer, a ‘u’ represents unsigned integer, ‘f’ represents floating point, ‘c’ represents complex floating point, ‘S’ represents 8-bit character string, ‘U’ represents 32-bit/character unicode string, and ‘V’ represents arbitrary.

`char type`
A traditional character code indicating the data type.

`char byteorder`
A character indicating the byte-order: ‘>’ (big-endian), ‘<’ (little- endian), ‘=’ (native), ‘|’ (irrelevant, ignore). All built-in data types have byteorder ‘=’.

`char hasobject`
A data-type bit-flag that determines if the data-type exhibits object- array like behavior. Each bit in this member is a flag which are named as:

- `NPY_ITEM_REFCOUNT` 
- `NPY_ITEM_HASOBJECT` 
- `NPY_ITEM_LISTPICKLE` 
- `NPY_ITEM_IS_POINTER` 
- `NPY_NEEDS_INIT` 
- `NPY_NEEDS_PYAPI` 
- `NPY_USE_GETITEM` 
- `NPY_USE_SETITEM` 
- `NPY_FROM_FIELDS` 
- `NPY_OBJECT_DTYPE_FLAGS`

`PyDataType_FLAGCHK(PyArray_Descr *dtype, int flags)`
Return true if all the given flags are set for the data-type object.

`PyDataType_REFCHK(PyArray_Descr *dtype)`
Equivalent to `PyDataType_FLAGCHK(dtype, NPY_ITEM_REFCOUNT)`. 
int type_num
A number that uniquely identifies the data type. For new data-types, this number is assigned when the data-type
is registered.

int elsize
For data types that are always the same size (such as long), this holds the size of the data type. For flexible data
types where different arrays can have a different elementsize, this should be 0.

int alignment
A number providing alignment information for this data type. Specifically, it shows how far from the start
of a 2-element structure (whose first element is a char), the compiler places an item of this type:
offsetof(struct {char c; type v;}, v)

PyArray_ArrayDescr subarray
If this is non-NULL, then this data-type descriptor is a C-style contiguous array of another data-type descriptor.
In other-words, each element that this descriptor describes is actually an array of some other base descriptor.
This is most useful as the data-type descriptor for a field in another data-type descriptor. The fields member
should be NULL if this is non-NULL (the fields member of the base descriptor can be non-NULL however).
The PyArray_ArrayDescr structure is defined using

typedef struct
{
    PyArray_Descr *base;
    PyObject *shape;
} PyArray_ArrayDescr;

The elements of this structure are:

PyArray_Descr base
    The data-type-descriptor object of the base-type.

PyObject shape
    The shape (always C-style contiguous) of the sub-array as a Python tuple.

PyObject fields
    If this is non-NULL, then this data-type-descriptor has fields described by a Python dictionary whose keys
    are names (and also titles if given) and whose values are tuples that describe the fields. Recall that a
data-type-descriptor always describes a fixed-length set of bytes. A field is a named sub-region of that total,
fixed-length collection. A field is described by a tuple composed of another data-type descriptor and a byte
offset. Optionally, the tuple may contain a title which is normally a Python string. These tuples are placed in
this dictionary keyed by name (and also title if given).

PyArray_ArrFuncs f
    A pointer to a structure containing functions that the type needs to implement internal features. These functions
are not the same thing as the universal functions (ufuncs) described later. Their signatures can vary arbitrarily.

PyArray_ArrFuncs
Functions implementing internal features. Not all of these function pointers must be defined for a given type.
The required members are nonzero, copyswap, copyswapn, setitem, getitem, and cast. These
are assumed to be non-NULL and NULL entries will cause a program crash. The other functions may be NULL
which will just mean reduced functionality for that data-type. (Also, the nonzero function will be filled in with
a default function if it is NULL when you register a user-defined data-type).

typedef struct
{
    PyArray_VectorUnaryFunc *cast[PyArray_NTYPES];
    PyArray_GetItemFunc *getitem;
    PyArray_SetItemFunc *setitem;
    PyArray_CopySwapNFunc *copyswapn;
    PyArray_CopySwapFunc *copyswap;
}
The concept of a behaved segment is used in the description of the function pointers. A behaved segment is one that is aligned and in native machine byte-order for the data-type. The nonzero, copyswap, copyswapn, getitem, and setitem functions can (and must) deal with misbehaved arrays. The other functions require behaved memory segments.

**void cast**

An array of function pointers to cast from the current type to all of the other builtin types. Each function casts a contiguous, aligned, and notswapped buffer pointed at by from to a contiguous, aligned, and notswapped buffer pointed at by to. The number of items to cast is given by n, and the arguments fromarr and toarr are interpreted as PyArrayObjects for flexible arrays to get itemsize information.

**PyObject getitem**

A pointer to a function that returns a standard Python object from a single element of the array object arr pointed to by data. This function must be able to deal with “misbehaved” (misaligned and/or swapped) arrays correctly.

**int setitem**

A pointer to a function that sets the Python object item into the array, arr, at the position pointed to by data. This function deals with “misbehaved” arrays. If successful, a zero is returned, otherwise, a negative one is returned (and a Python error set).

**void copyswapn**

**void copyswap**

These members are both pointers to functions to copy data from src to dest and swap if indicated. The value of arr is only used for flexible (NPY_STRING, NPY_UNICODE, and NPY_VOID) arrays (and is obtained from arr->descr->elsize). The second function copies a single value, while the first loops over n values with the provided strides. These functions can deal with misbehaved src data. If src is NULL then no copy is performed. If swap is 0, then no byteswapping occurs. It is assumed that dest and src do not overlap. If they overlap, then use memmove(...) first followed by copyswap(n) with NULL valued src.

**int compare**

A pointer to a function that compares two elements of the array, arr, pointed to by d1 and d2. This function requires behaved arrays. The return value is 1 if *d1 > *d2, 0 if *d1 == *d2, and -1 if *d1 < *d2. The array object arr is used to retrieve itemsize and field information for flexible arrays.

**int argmax**

A pointer to a function that retrieves the index of the largest of n elements in arr beginning at the element pointed to by data. This function requires that the memory segment be contiguous and behaved.
return value is always 0. The index of the largest element is returned in max_ind.

**void dotfunc**
A pointer to a function that multiplies two n-length sequences together, adds them, and places the result in element pointed to by op of arr. The start of the two sequences are pointed to by ip1 and ip2. To get to the next element in each sequence requires a jump of is1 and is2 bytes, respectively. This function requires behaved (though not necessarily contiguous) memory.

**int scanfunc**
A pointer to a function that scans (scanf style) one element of the corresponding type from the file descriptor fd into the array memory pointed to by ip. The array is assumed to be behaved. If sep is not NULL, then a separator string is also scanned from the file before returning. The last argument arr is the array to be scanned into. A 0 is returned if the scan is successful. A negative number indicates something went wrong: -1 means the end of file was reached before the separator string could be scanned, -4 means that the end of file was reached before the element could be scanned, and -3 means that the element could not be interpreted from the format string. Requires a behaved array.

**int fromstr**
A pointer to a function that converts the string pointed to by str to one element of the corresponding type and places it in the memory location pointed to by ip. After the conversion is completed, *endptr points to the rest of the string. The last argument arr is the array into which ip points (needed for variable-size data types). Returns 0 on success or -1 on failure. Requires a behaved array.

**Bool nonzero**
A pointer to a function that returns TRUE if the item of arr pointed to by data is nonzero. This function can deal with misbehaved arrays.

**void fill**
A pointer to a function that fills a contiguous array of given length with data. The first two elements of the array must already be filled-in. From these two values, a delta will be computed and the values from item 3 to the end will be computed by repeatedly adding this computed delta. The data buffer must be well-behaved.

**void fillwithscalar**
A pointer to a function that fills a contiguous buffer of the given length with a single scalar value whose address is given. The final argument is the array which is needed to get the itemsize for variable-length arrays.

**int sort**
An array of function pointers to a particular sorting algorithms. A particular sorting algorithm is obtained using a key (so far PyArray_QUICKSORT, PyArray_HEAPSORT, and PyArray_MERGESORT are defined). These sorts are done in-place assuming contiguous and aligned data.

**int argsort**
An array of function pointers to sorting algorithms for this data type. The same sorting algorithms as for sort are available. The indices producing the sort are returned in result (which must be initialized with indices 0 to length-1 inclusive).

**PyObject castdict**
Either NULL or a dictionary containing low-level casting functions for user-defined data-types. Each function is wrapped in a PyObject * and keyed by the data-type number.

**PyArray_SCALARKIND scalarkind**
A function to determine how scalars of this type should be interpreted. The argument is NULL or a 0-dimensional array containing the data (if that is needed to determine the kind of scalar). The return value must be of type PyArray_SCALARKIND.

**int cancastscalarkindto**
Either NULL or an array of PyArray_NSCALARKINDS pointers. These pointers should each be either...
NULL or a pointer to an array of integers (terminated by PyArray_NOTYPE) indicating data-types that a scalar of this data-type of the specified kind can be cast to safely (this usually means without losing precision).

int cancastto

Either NULL or an array of integers (terminated by PyArray_NOTYPE ) indicated data-types that this data-type can be cast to safely (this usually means without losing precision).

int listpickle

Unused.

The PyArray_Type typeobject implements many of the features of Python objects including the tp_as_number, tp_as_sequence, tp_as_mapping, and tp_as_buffer interfaces. The rich comparison (tp_richcompare) is also used along with new-style attribute lookup for methods (tp_methods) and properties (tp_getset). The PyArray_Type can also be sub-typed.

Tip: The tp_as_number methods use a generic approach to call whatever function has been registered for handling the operation. The function PyNumeric_SetOps(..) can be used to register functions to handle particular mathematical operations (for all arrays). When the umath module is imported, it sets the numeric operations for all arrays to the corresponding ufuncs. The tp_str and tp_repr methods can also be altered using PyString_SetStringFunction(...).

PyUFunc_Type

PyUFunc_Type

The ufunc object is implemented by creation of the PyUFunc_Type. It is a very simple type that implements only basic getattribute behavior, printing behavior, and has call behavior which allows these objects to act like functions. The basic idea behind the ufunc is to hold a reference to fast 1-dimensional (vector) loops for each data type that supports the operation. These one-dimensional loops all have the same signature and are the key to creating a new ufunc. They are called by the generic looping code as appropriate to implement the N-dimensional function. There are also some generic 1-d loops defined for floating and complexfloating arrays that allow you to define a ufunc using a single scalar function (e.g. atanh).

PyUFuncObject

The core of the ufunc is the PyUFuncObject which contains all the information needed to call the underlying C-code loops that perform the actual work. It has the following structure:

typedef struct {  
    PyObject_HEAD  
    int nin;  
    int nout;  
    int nargs;  
    int identity;  
    PyUFuncGenericFunction *functions;  
    void **data;  
    int ntypes;  
    int check_return;  
    char *name;  
    char *types;  
    char *doc;  
    void *ptr;  
    PyObject *obj;  
    PyObject *userloops;  
} PyUFuncObject;

PyObject_HEAD

required for all Python objects.
int \texttt{nin}
\begin{itemize}
\item The number of input arguments.
\end{itemize}

int \texttt{nout}
\begin{itemize}
\item The number of output arguments.
\end{itemize}

int \texttt{nargs}
\begin{itemize}
\item The total number of arguments \((\texttt{nin} + \texttt{nout})\). This must be less than \texttt{NPY_MAXARGS}.
\end{itemize}

int \texttt{identity}
\begin{itemize}
\item Either \texttt{PyUFunc\_One}, \texttt{PyUFunc\_Zero}, or \texttt{PyUFunc\_None} to indicate the identity for this operation. It is only used for a reduce-like call on an empty array.
\end{itemize}

\textbf{void \texttt{functions}}
\begin{itemize}
\item An array of function pointers — one for each data type supported by the ufunc. This is the vector loop that is called to implement the underlying function \texttt{dims} \([0]\) times. The first argument, \texttt{args}, is an array of \texttt{nargs} pointers to behaved memory. Pointers to the data for the input arguments are first, followed by the pointers to the data for the output arguments. How many bytes must be skipped to get to the next element in the sequence is specified by the corresponding entry in the \texttt{steps} array. The last argument allows the loop to receive extra information. This is commonly used so that a single, generic vector loop can be used for multiple functions. In this case, the actual scalar function to call is passed in as \texttt{extradata}. The size of this function pointer array is \texttt{ntypes}.
\end{itemize}

\textbf{void \texttt{data}}
\begin{itemize}
\item Extra data to be passed to the 1-d vector loops or \texttt{NULL} if no extra-data is needed. This C-array must be the same size (\textit{i.e.} \texttt{ntypes}) as the functions array. \texttt{NULL} is used if \texttt{extra\_data} is not needed. Several C-API calls for UFuncs are just 1-d vector loops that make use of this extra data to receive a pointer to the actual function to call.
\end{itemize}

int \texttt{ntypes}
\begin{itemize}
\item The number of supported data types for the ufunc. This number specifies how many different 1-d loops (of the builtin data types) are available.
\end{itemize}

int \texttt{check\_return}
\begin{itemize}
\item Obsolete and unused. However, it is set by the corresponding entry in the main ufunc creation routine: \texttt{PyUFunc\_FromFuncAndData(...)}.
\end{itemize}

char \texttt{name}
\begin{itemize}
\item A string name for the ufunc. This is used dynamically to build the \texttt{\_doc\_} attribute of ufuncs.
\end{itemize}

char \texttt{types}
\begin{itemize}
\item An array of \texttt{nargs} \times \texttt{ntypes} 8-bit type\_numbers which contains the type signature for the function for each of the supported (builtin) data types. For each of the \texttt{ntypes} functions, the corresponding set of type numbers in this array shows how the \texttt{args} argument should be interpreted in the 1-d vector loop. These type numbers do not have to be the same type and mixed-type ufuncs are supported.
\end{itemize}

char \texttt{doc}
\begin{itemize}
\item Documentation for the ufunc. Should not contain the function signature as this is generated dynamically when \texttt{\_doc\_} is retrieved.
\end{itemize}

\textbf{void \texttt{ptr}}
\begin{itemize}
\item Any dynamically allocated memory. Currently, this is used for dynamic ufuncs created from a python function to store room for the types, data, and name members.
\end{itemize}

\textbf{PyObject \texttt{obj}}
\begin{itemize}
\item For ufuncs dynamically created from python functions, this member holds a reference to the underlying Python function.
\end{itemize}

\textbf{PyObject \texttt{userloops}}
\begin{itemize}
\item A dictionary of user-defined 1-d vector loops (stored as CObject ptrs) for user-defined types. A loop may
be registered by the user for any user-defined type. It is retrieved by type number. User defined type numbers are always larger than NPY_USERDEF.

**PyArrayIter_Type**

This is an iterator object that makes it easy to loop over an N-dimensional array. It is the object returned from the flat attribute of an ndarray. It is also used extensively throughout the implementation internals to loop over an N-dimensional array. The ip_as_mapping interface is implemented so that the iterator object can be indexed (using 1-d indexing), and a few methods are implemented through the tp_methods table. This object implements the next method and can be used anywhere an iterator can be used in Python.

**PyArrayIterObject**

The C-structure corresponding to an object of PyArrayIter_Type is the PyArrayIterObject. The PyArrayIterObject is used to keep track of a pointer into an N-dimensional array. It contains associated information used to quickly march through the array. The pointer can be adjusted in three basic ways: 1) advance to the “next” position in the array in a C-style contiguous fashion, 2) advance to an arbitrary N-dimensional coordinate in the array, and 3) advance to an arbitrary one-dimensional index into the array. The members of the PyArrayIterObject structure are used in these calculations. Iterator objects keep their own dimension and strides information about an array. This can be adjusted as needed for “broadcasting,” or to loop over only specific dimensions.

typedef struct {
    PyObject_HEAD
    int nd_m1;
    npy_intp index;
    npy_intp size;
    npy_intp coordinates[NPY_MAXDIMS];
    npy_intp dims_m1[NPY_MAXDIMS];
    npy_intp strides[NPY_MAXDIMS];
    npy_intp backstrides[NPY_MAXDIMS];
    npy_intp factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char *dataptr;
    Bool contiguous;
} PyArrayIterObject;

int nd_m1
   \( N - 1 \) where \( N \) is the number of dimensions in the underlying array.

npy_intp index
   The current 1-d index into the array.

npy_intp size
   The total size of the underlying array.

npy_intp coordinates
   An \( N \)-dimensional index into the array.

npy_intp dims_m1
   The size of the array minus 1 in each dimension.

npy_intp strides
   The strides of the array. How many bytes needed to jump to the next element in each dimension.

npy_intp backstrides
   How many bytes needed to jump from the end of a dimension back to its beginning. Note that backstrides \([k]=strides[k]\times dims_m1[k]\), but it is stored here as an optimization.
npy_intp factors
    This array is used in computing an N-d index from a 1-d index. It contains needed products of the dimensions.

PyArrayObject ao
    A pointer to the underlying ndarray this iterator was created to represent.

char dataptr
    This member points to an element in the ndarray indicated by the index.

bool contiguous
    This flag is true if the underlying array is NPY_C_CONTIGUOUS. It is used to simplify calculations when possible.

How to use an array iterator on a C-level is explained more fully in later sections. Typically, you do not need to concern yourself with the internal structure of the iterator object, and merely interact with it through the use of the macros PyArray_ITER_NEXT (it), PyArray_ITER_GOTO (it, dest), or PyArray_ITER_GOTO1D (it, index). All of these macros require the argument it to be a PyArrayIterObject *.

PyArrayMultiIter_Type

PyArrayMultiIter_Type
    This type provides an iterator that encapsulates the concept of broadcasting. It allows N arrays to be broadcast together so that the loop progresses in C-style contiguous fashion over the broadcasted array. The corresponding C-structure is the PyArrayMultiIterObject whose memory layout must begin any object, obj, passed in to the PyArray_Broadcast (obj) function. Broadcasting is performed by adjusting array iterators so that each iterator represents the broadcasted shape and size, but has its strides adjusted so that the correct element from the array is used at each iteration.

PyArrayMultiIterObject

typedef struct {
    PyObject_HEAD
    int numiter;
    npy_intp size;
    npy_intp index;
    int nd;
    npy_intp dimensions[NPY_MAXDIMS];
    PyArrayIterObject *iters[NPY_MAXDIMS];
} PyArrayMultiIterObject;

PyObject_HEAD
    Needed at the start of every Python object (holds reference count and type identification).

int numiter
    The number of arrays that need to be broadcast to the same shape.

npy_intp size
    The total broadcasted size.

npy_intp index
    The current (1-d) index into the broadcasted result.

int nd
    The number of dimensions in the broadcasted result.

npy_intp dimensions
    The shape of the broadcasted result (only nd slots are used).
**PyArrayIterObject** *iters*

An array of iterator objects that holds the iterators for the arrays to be broadcast together. On return, the iterators are adjusted for broadcasting.

**PyArrayNeighborhoodIter_Type**

This is an iterator object that makes it easy to loop over an N-dimensional neighborhood.

**PyArrayNeighborhoodIterObject**

The C-structure corresponding to an object of *PyArrayNeighborhoodIter_Type* is the *PyArrayNeighborhoodIterObject*.

**PyArrayFlags_Type**

When the flags attribute is retrieved from Python, a special builtin object of this type is constructed. This special type makes it easier to work with the different flags by accessing them as attributes or by accessing them as if the object were a dictionary with the flag names as entries.

**ScalarArrayTypes**

There is a Python type for each of the different built-in data types that can be present in the array. Most of these are simple wrappers around the corresponding data type in C. The C-names for these types are *Py(TYPE)ArrType_Type* where *TYPE* can be


These type names are part of the C-API and can therefore be created in extension C-code. There is also a *PyIntpArrType_Type* and a *PyUIntpArrType_Type* that are simple substitutes for one of the integer types that can hold a pointer on the platform. The structure of these scalar objects is not exposed to C-code. The function *PyArray_ScalarAsCtype(..)* can be used to extract the C-type value from the array scalar and the function *PyArray_Scalar(...)* can be used to construct an array scalar from a C-value.

### 5.1.2 Other C-Structures

A few new C-structures were found to be useful in the development of NumPy. These C-structures are used in at least one C-API call and are therefore documented here. The main reason these structures were defined is to make it easy to use the Python ParseTuple C-API to convert from Python objects to a useful C-Object.

**PyArray_Dims**

This structure is very useful when shape and/or strides information is supposed to be interpreted. The structure is:

```c
typedef struct {
    npy_intp *ptr;
    int len;
} PyArray_Dims;
```
The members of this structure are

```c
npy_intp ptr
    A pointer to a list of (npy_intp) integers which usually represent array shape or array strides.

int len
    The length of the list of integers. It is assumed safe to access ptr[0] to ptr[len-1].
```

**PyArray_Chunk**

This is equivalent to the buffer object structure in Python up to the ptr member. On 32-bit platforms \(\text{i.e. if NPY_SIZEOF_INT == NPY_SIZEOF_INTP}\) or in Python 2.5, the len member also matches an equivalent member of the buffer object. It is useful to represent a generic single-segment chunk of memory.

```c
typedef struct {
    PyObject_HEAD
    PyObject *base;
    void *ptr;
    npy_intp len;
    int flags;
} PyArray_Chunk;
```

The members are

**PyObject\_HEAD**

Necessary for all Python objects. Included here so that the PyArray\_Chunk structure matches that of the buffer object (at least to the len member).

**PyObject base**

The Python object this chunk of memory comes from. Needed so that memory can be accounted for properly.

**void ptr**

A pointer to the start of the single-segment chunk of memory.

**npy_intp len**

The length of the segment in bytes.

**int flags**

Any data flags (\text{e.g. NPY\_WRITEABLE}) that should be used to interpret the memory.

**PyArrayInterface**

See Also:

*The Array Interface*

**PyArrayInterface**

The PyArrayInterface structure is defined so that NumPy and other extension modules can use the rapid array interface protocol. The __array_struct__ method of an object that supports the rapid array interface protocol should return a PyObject that contains a pointer to a PyArrayInterface structure with the relevant details of the array. After the new array is created, the attribute should be DECREP’d which will free the PyArrayInterface structure. Remember to INCREF the object (whose __array_struct__ attribute was retrieved) and point the base member of the new PyArrayObject to this same object. In this way the memory for the array will be managed correctly.
```c
typedef struct {
    int two;
    int nd;
    char typekind;
    int itemsize;
    int flags;
    npy_intp *shape;
    npy_intp *strides;
    void *data;
    PyObject *descr;
} PyArrayInterface;
```

- `two`: the integer 2 as a sanity check.
- `nd`: the number of dimensions in the array.
- `itemsize`: The number of bytes each item in the array requires.
- `flags`: Any of the bits `NPY_C_CONTIGUOUS` (1), `NPY_F_CONTIGUOUS` (2), `NPY_ALIGNED` (0x100), `NPY_NOTSWAPPED` (0x200), or `NPY_WRITEABLE` (0x400) to indicate something about the data. The `NPY_ALIGNED`, `NPY_C_CONTIGUOUS`, and `NPY_F_CONTIGUOUS` flags can actually be determined from the other parameters. The flag `NPY_ARR_HAS_DESCR` (0x800) can also be set to indicate to objects consuming the version 3 array interface that the descr member of the structure is present (it will be ignored by objects consuming version 2 of the array interface).
- `shape`: An array containing the size of the array in each dimension.
- `strides`: An array containing the number of bytes to jump to get to the next element in each dimension.
- `data`: A pointer to the first element of the array.
- `descr`: A Python object describing the data-type in more detail (same as the `descr` key in `__array_interface__`). This can be `NULL` if `typekind` and `itemsize` provide enough information. This field is also ignored unless `ARR_HAS_DESCR` flag is on in `flags`.

**Internally used structures**

Internally, the code uses some additional Python objects primarily for memory management. These types are not accessible directly from Python, and are not exposed to the C-API. They are included here only for completeness and assistance in understanding the code.

**PyUFuncLoopObject**

A loose wrapper for a C-structure that contains the information needed for looping. This is useful if you are trying to understand the ufunc looping code. The `PyUFuncLoopObject` is the associated C-structure. It is defined in the `ufuncobject.h` header.
**PyUFuncReduceObject**
A loose wrapper for the C-structure that contains the information needed for reduce-like methods of ufuncs. This is useful if you are trying to understand the reduce, accumulate, and reduce-at code. The *PyUFuncReduceObject* is the associated C-structure. It is defined in the *ufuncobject.h* header.

**PyUFunc_Loop1d**
A simple linked-list of C-structures containing the information needed to define a 1-d loop for a ufunc for every defined signature of a user-defined data-type.

**PyArrayMapIter_Type**
Advanced indexing is handled with this Python type. It is simply a loose wrapper around the C-structure containing the variables needed for advanced array indexing. The associated C-structure, *PyArrayMapIterObject*, is useful if you are trying to understand the advanced-index mapping code. It is defined in the *arrayobject.h* header. This type is not exposed to Python and could be replaced with a C-structure. As a Python type it takes advantage of reference-counted memory management.

### 5.2 System configuration

When NumPy is built, information about system configuration is recorded, and is made available for extension modules using Numpy’s C API. These are mostly defined in *numpyconfig.h* (included in *ndarrayobject.h*). The public symbols are prefixed by `NPY_*`. Numpy also offers some functions for querying information about the platform in use.

For private use, Numpy also constructs a *config.h* in the NumPy include directory, which is not exported by Numpy (that is a python extension which use the numpy C API will not see those symbols), to avoid namespace pollution.

#### 5.2.1 Data type sizes

The *NPY_SIZEOF_*(CTYPE) constants are defined so that sizeof information is available to the pre-processor.

```
NPY_SIZEOF_SHORT
    sizeof(short)

NPY_SIZEOF_INT
    sizeof(int)

NPY_SIZEOF_LONG
    sizeof(long)

NPY_SIZEOF_LONG_LONG
    sizeof(longlong) where longlong is defined appropriately on the platform (A macro defines NPY_SIZEOF_LONGLONG as well.)

NPY_SIZEOF_PY_LONG_LONG

NPY_SIZEOF_FLOAT
    sizeof(float)

NPY_SIZEOF_DOUBLE
    sizeof(double)

NPY_SIZEOF_LONG_DOUBLE
    sizeof(longdouble) (A macro defines NPY_SIZEOF_LONDDOUBLE as well.)

NPY_SIZEOF_PY_INTPTR_T
    Size of a pointer on this platform (sizeof(void *)) (A macro defines NPY_SIZEOF_INTP as well.)
```
5.2.2 Platform information

NPY_CPU_X86

NPY_CPU_AMD64

NPY_CPU_IA64

NPY_CPU_PPC

NPY_CPU_PPC64

NPY_CPU_SPARC

NPY_CPU_SPARC64

NPY_CPU_S390

NPY_CPU_PARISC
   New in version 1.3.0. CPU architecture of the platform; only one of the above is defined.
   Defined in numpy/npy_cpu.h

NPY_LITTLE_ENDIAN

NPY_BIG_ENDIAN

NPY_BYTE_ORDER
   New in version 1.3.0. Portable alternatives to the endian.h macros of GNU Libc. If big endian,
   NPY_BYTE_ORDER == NPY_BIG_ENDIAN, and similarly for little endian architectures.
   Defined in numpy/npy_endian.h

PyArray_GetEndianness()
   New in version 1.3.0. Returns the endianness of the current platform. One of NPY_CPU_BIG,
   NPY_CPU_LITTLE, or NPY_CPU_UNKNOWN_ENDIAN.

5.3 Data Type API

The standard array can have 21 different data types (and has some support for adding your own types). These data
types all have an enumerated type, an enumerated type-character, and a corresponding array scalar Python type object
placed in a hierarchy). There are also standard C typedefs to make it easier to manipulate elements of the given data
type. For the numeric types, there are also bit-width equivalent C typedefs and named typenumbers that make it easier
to select the precision desired.

Warning: The names for the types in c code follows c naming conventions more closely. The Python names for
these types follow Python conventions. Thus, NPY_FLOAT picks up a 32-bit float in C, but numpy.float_ in
Python corresponds to a 64-bit double. The bit-width names can be used in both Python and C for clarity.
5.3.1 Enumerated Types

There is a list of enumerated types defined providing the basic 21 data types plus some useful generic names. Whenever the code requires a type number, one of these enumerated types is requested. The types are all called \texttt{NPY_{NAME}} where \texttt{(NAME)} can be

\begin{verbatim}
BOOL, BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG, ULONG, LONGLONG, ULONG-
LONG, FLOAT, DOUBLE, LONDBLUE, CFLOAT, CDOUBLE, CLONDBLUE, OBJECT,
STRING, UNICODE, VOID
\end{verbatim}

\texttt{NTYPES, NOTYPE, USERDEF, DEFAULT_TYPE}

The various character codes indicating certain types are also part of an enumerated list. References to type characters (should they be needed at all) should always use these enumerations. The form of them is \texttt{NPY_{NAME}LTR} where \texttt{(NAME)} can be

\begin{verbatim}
BOOL, BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG, ULONG, LONGLONG, ULONG-
LONG, FLOAT, DOUBLE, LONDBLUE, CFLOAT, CDOUBLE, CLONDBLUE, OBJECT,
STRING, VOID
\end{verbatim}

\texttt{INTP, UINTP}

\texttt{GENBOOL, SIGNED, UNSIGNED, FLOATING, COMPLEX}

The latter group of \texttt{(NAME)s} corresponds to letters used in the array interface \texttt{typestring} specification.

5.3.2 Defines

Max and min values for integers

\begin{verbatim}
NPY_MAX_INT{bits}

NPY_MAX_UINT{bits}

NPY_MIN_INT{bits}
\end{verbatim}

These are defined for \texttt{(bits)} = 8, 16, 32, 64, 128, and 256 and provide the maximum (minimum) value of the corresponding (unsigned) integer type. Note: the actual integer type may not be available on all platforms (i.e. 128-bit and 256-bit integers are rare).

\begin{verbatim}
NPY_MIN_{type}
This is defined for \texttt{(type)} = BYTE, SHORT, INT, LONG, LONGLONG, INTP

NPY_MAX_{type}
This is defined for all defined for \texttt{(type)} = BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG,
ULONG, LONGLONG, ULLONG, INTP, UINTP
\end{verbatim}

Number of bits in data types

All \texttt{NPY_SIZEOF\_\{CTYPE\}} constants have corresponding \texttt{NPY_BITSOF\_\{CTYPE\}} constants defined. The \texttt{NPY_BITSOF\_\{CTYPE\}} constants provide the number of bits in the data type. Specifically, the available \texttt{\{CTYPE\}s} are

\begin{verbatim}
BOOL, CHAR, SHORT, INT, LONG, LONGLONG, FLOAT, DOUBLE, LONDBLUE
\end{verbatim}
Bit-width references to enumerated typenums

All of the numeric data types (integer, floating point, and complex) have constants that are defined to be a specific enumerated type number. Exactly which enumerated type a bit-width type refers to is platform dependent. In particular, the constants available are `PyArray_{NAME}{BITS}` where `{NAME}` is `INT`, `UINT`, `FLOAT`, `COMPLEX` and `{BITS}` can be 8, 16, 32, 64, 80, 96, 128, 160, 192, 256, and 512. Obviously not all bit-widths are available on all platforms for all the kinds of numeric types. Commonly 8-, 16-, 32-, 64-bit integers; 32-, 64-bit floats; and 64-, 128-bit complex types are available.

Integer that can hold a pointer

The constants `PyArray_INTP` and `PyArray_UINTP` refer to an enumerated integer type that is large enough to hold a pointer on the platform. Index arrays should always be converted to `PyArray_INTP`, because the dimension of the array is of type `npy_intp`.

5.3.3 C-type names

There are standard variable types for each of the numeric data types and the bool data type. Some of these are already available in the C-specification. You can create variables in extension code with these types.

Boolean

```python
npy_bool
```

`unsigned char`; The constants `NPY_FALSE` and `NPY_TRUE` are also defined.

(Un)Signed Integer

Unsigned versions of the integers can be defined by pre-pending a ‘u’ to the front of the integer name.

```python
npy_(u)byte
    (unsigned) char
npy_(u)short
    (unsigned) short
npy_(u)int
    (unsigned) int
npy_(u)long
    (unsigned) long int
npy_(u)longlong
    (unsigned long long int)
npy_(u)intp
    (unsigned) Py_intptr_t (an integer that is the size of a pointer on the platform).
```

(Complex) Floating point

```python
npy_(c)float
    float
npy_(c)double
    double
```
complex types are structures with .real and .imag members (in that order).

Bit-width names

There are also typedefs for signed integers, unsigned integers, floating point, and complex floating point types of specific bit-widths. The available type names are

\[ \text{npy}_\text{int}\{\text{bits}\}, \text{npy}_\text{uint}\{\text{bits}\}, \text{npy}_\text{float}\{\text{bits}\}, \text{and} \text{npy}_\text{complex}\{\text{bits}\} \]

where \{bits\} is the number of bits in the type and can be 8, 16, 32, 64, 128, and 256 for integer types; 16, 32, 64, 80, 96, 128, and 256 for floating-point types; and 32, 64, 128, 160, 192, and 512 for complex-valued types. Which bit-widths are available is platform dependent. The bolded bit-widths are usually available on all platforms.

5.3.4 Printf Formatting

For help in printing, the following strings are defined as the correct format specifier in printf and related commands.

\[ \text{NPY}_\text{LONGLONG_FMT}, \text{NPY}_\text{ULONGLONG_FMT}, \text{NPY}_\text{INTP_FMT}, \text{NPY}_\text{UINTP_FMT}. \]

5.4 Array API

The test of a first-rate intelligence is the ability to hold two opposed ideas in the mind at the same time, and still retain the ability to function.
— F. Scott Fitzgerald

For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled.
— Richard P. Feynman

5.4.1 Array structure and data access

These macros all access the PyArrayObject structure members. The input argument, obj, can be any PyObject * that is directly interpretable as a PyArrayObject * (any instance of the PyArray_Type and its sub-types).

\[ \text{void} * \text{PyArray\_DATA}(\text{PyObject} *\text{obj}) \]

\[ \text{char} * \text{PyArray\_BYTES}(\text{PyObject} *\text{obj}) \]

These two macros are similar and obtain the pointer to the data-buffer for the array. The first macro can (and should) be assigned to a particular pointer where the second is for generic processing. If you have not guaranteed a contiguous and/or aligned array then be sure you understand how to access the data in the array to avoid memory and/or alignment problems.

\[ \text{npy\_intp} * \text{PyArray\_DIMS}(\text{PyObject} *\text{arr}) \]

\[ \text{npy\_intp} * \text{PyArray\_STRIDES}(\text{PyObject} *\text{arr}) \]

\[ \text{npy\_intp} \text{PyArray\_DIM}(\text{PyObject} *\text{arr}, \text{int} \text{n}) \]

Return the shape in the \text{n} \text{th} dimension.
npy_intp PyArray_STRIDE (PyObject* arr, int n)
    Return the stride in the n\textsuperscript{th} dimension.

PyObject* PyArray_BASE (PyObject* arr)

PyArray_Descr* PyArray_DESCR (PyObject* arr)

int PyArray_FLAGS (PyObject* arr)

int PyArray_ITEMSIZE (PyObject* arr)
    Return the itemsize for the elements of this array.

int PyArray_TYPE (PyObject* arr)
    Return the (builtin) typenumber for the elements of this array.

PyObject* PyArray_GETITEM (PyObject* arr, void* itemptr)
    Get a Python object from the ndarray, arr, at the location pointed to by itemptr. Return NULL on failure.

int PyArray_SETITEM (PyObject* arr, void* itemptr, PyObject* obj)
    Convert obj and place it in the ndarray, arr, at the place pointed to by itemptr. Return -1 if an error occurs or 0 on success.

npy_intp PyArray_SIZE (PyObject* arr)
    Returns the total size (in number of elements) of the array.

npy_intp PyArray_Size (PyObject* obj)
    Returns 0 if obj is not a sub-class of bigndarray. Otherwise, returns the total number of elements in the array.
    Safer version of PyArray_SIZE (obj).

npy_intp PyArray_NBYTES (PyObject* arr)
    Returns the total number of bytes consumed by the array.

Data access

These functions and macros provide easy access to elements of the ndarray from C. These work for all arrays. You may need to take care when accessing the data in the array, however, if it is not in machine byte-order, misaligned, or not writable. In other words, be sure to respect the state of the flags unless you know what you are doing, or have previously guaranteed an array that is writeable, aligned, and in machine byte-order using PyArray_FromAny. If you wish to handle all types of arrays, the copyswap function for each type is useful for handling misbehaved arrays. Some platforms (e.g. Solaris) do not like misaligned data and will crash if you de-reference a misaligned pointer. Other platforms (e.g. x86 Linux) will just work more slowly with misaligned data.

void* PyArray_GetPtr (PyArrayObject* aobj, npy_intp* ind)
    Return a pointer to the data of the ndarray, aobj, at the N-dimensional index given by the c-array, ind, (which must be at least aobj->nd in size). You may want to typecast the returned pointer to the data type of the ndarray.

void* PyArray_GETPTR1 (PyObject* obj, <npy_intp> i)

void* PyArray_GETPTR2 (PyObject* obj, <npy_intp> i, <npy_intp> j)

void* PyArray_GETPTR3 (PyObject* obj, <npy_intp> i, <npy_intp> j, <npy_intp> k)

void* PyArray_GETPTR4 (PyObject* obj, <npy_intp> i, <npy_intp> j, <npy_intp> k, <npy_intp> l)
    Quick, inline access to the element at the given coordinates in the ndarray, obj, which must have respectively 1,
2, 3, or 4 dimensions (this is not checked). The corresponding \(i, j, k,\) and \(l\) coordinates can be any integer but will be interpreted as \texttt{npy\_intp}. You may want to typecast the returned pointer to the data type of the ndarray.

### 5.4.2 Creating arrays

#### From scratch

\begin{verbatim}
PyObject* PyArray_NewFromDescr (PyTypeObject* subtype, PyArray_Descr* descr, int nd, npy_intp* dims, npy_intp* strides, void* data, int flags, PyObject* obj)
This is the main array creation function. Most new arrays are created with this flexible function. The returned object is an object of Python-type \texttt{subtype}, which must be a subtype of \texttt{PyArray\_Type}. The array has \(nd\) dimensions, described by \texttt{dims}. The data-type descriptor of the new array is \texttt{descr}. If \texttt{subtype} is not &\texttt{PyArray\_Type} (e.g. a Python subclass of the ndarray), then \texttt{obj} is the object to pass to the \texttt{\_array\_finalize\_} method of the subclass. If \texttt{data} is NULL, then new memory will be allocated and \texttt{flags} can be non-zero to indicate a Fortran-style contiguous array. If \texttt{data} is not NULL, then it is assumed to point to the memory to be used for the array and the \texttt{flags} argument is used as the new flags for the array (except the state of \texttt{NPY\_OWNDATA} and \texttt{UPDATEIFCOPY} flags of the new array will be reset). In addition, if \texttt{data} is non-NULL, then \texttt{strides} can also be provided. If \texttt{strides} is NULL, then the array strides are computed as C-style contiguous (default) or Fortran-style contiguous (\texttt{flags} is nonzero for \texttt{data} = NULL or flags & \texttt{NPY\_F\_CONTIGUOUS} is nonzero non-NULL data). Any provided \texttt{dims} and \texttt{strides} are copied into newly allocated dimension and strides arrays for the new array object.

PyObject* PyArray_New (PyTypeObject* subtype, int nd, npy_intp* dims, int type_num, npy_intp* strides, void* data, int itemsize, int flags, PyObject* obj)
This is similar to \texttt{PyArray\_DescrNew} (...) except you specify the data-type descriptor with \texttt{type_num} and \texttt{itemsize}, where \texttt{type_num} corresponds to a builtin (or user-defined) type. If the type always has the same number of bytes, then \texttt{itemsize} is ignored. Otherwise, \texttt{itemsize} specifies the particular size of this array.

\textbf{Warning: }If data is passed to \texttt{PyArray\_NewFromDescr} or \texttt{PyArray\_New}, this memory must not be deallocated until the new array is deleted. If this data came from another Python object, this can be accomplished using \texttt{Py\_INCREF} on that object and setting the base member of the new array to point to that object. If strides are passed in they must be consistent with the dimensions, the \texttt{itemsize}, and the data of the array.

PyObject* PyArray_SimpleNew (int nd, npy_intp* dims, int typenum)
Create a new uninitialized array of type, \texttt{typenum}, whose size in each of \(nd\) dimensions is given by the integer array, \texttt{dims}. This function cannot be used to create a flexible-type array (no \texttt{itemsize} given).

PyObject* PyArray_SimpleNewFromData (int nd, npy_intp* dims, int typenum, void* data)
Create an array wrapper around \texttt{data} pointed to by the given pointer. The array flags will have a default that the data area is well-behaved and C-style contiguous. The shape of the array is given by the \texttt{dims} c-array of length \(nd\). The data-type of the array is indicated by \texttt{typenum}.

PyObject* PyArray_SimpleNewFromDescr (int nd, npy_intp* dims, PyArray_Descr* descr)
Create a new array with the provided data-type descriptor, \texttt{descr}, of the shape determined by \texttt{nd} and \texttt{dims}.

\texttt{PyArray\_FILLWBYTE} (PyObject* obj, int val)
Fill the array pointed to by \texttt{obj} —which must be a (subclass of) \texttt{bigndarray}—with the contents of \texttt{val} (evaluated as a byte).

PyObject* PyArray_Zeros (int nd, npy_intp* dims, PyArray_Descr* dtype, int fortran)
Construct a new \(nd\) -dimensional array with shape given by \texttt{dims} and data type given by \texttt{dtype}. If \texttt{fortran} is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. Fill the memory with zeros (or the 0 object if \texttt{dtype} corresponds to \texttt{PyArray\_OBJECT}).

PyObject* PyArray\_ZEROS (int nd, npy_intp* dims, int type_num, int fortran)
Macro form of \texttt{PyArray\_Zeros} which takes a type-number instead of a data-type object.
\end{verbatim}
PyObject* `PyArray_Empty` (int nd, npy_intp* dims, PyArray_Descr* dtype, int fortran)
Construct a new `nd`-dimensional array with shape given by `dims` and data type given by `dtype`. If `fortran` is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. The array is uninitialized unless the data type corresponds to `PyArray_OBJECT` in which case the array is filled with `Py_None`.

PyObject* `PyArray_EMPTY` (int nd, npy_intp* dims, int typenum, int fortran)
Macro form of `PyArray_Empty` which takes a type-number, `typenum`, instead of a data-type object.

PyObject* `PyArray_Arange` (double start, double stop, double step, int typenum)
Construct a new 1-dimensional array of data-type, `typenum`, that ranges from `start` to `stop` (exclusive) in increments of `step`. Equivalent to `arange(start, stop, step, dtype)`.

PyObject* `PyArray_ArangeObj` (PyObject* start, PyObject* stop, PyObject* step, PyArray_Descr* descr)
Construct a new 1-dimensional array of data-type determined by `descr`, that ranges from `start` to `stop` (exclusive) in increments of `step`. Equivalent to `arange(start, stop, step, typenum)`.

From other objects

PyObject* `PyArray_FromAny` (PyObject* op, PyArray_Descr* dtype, int min_depth, int max_depth, int requirements, PyObject* context)
This is the main function used to obtain an array from any nested sequence, or object that exposes the array interface, `op`. The parameters allow specification of the required `dtype`, the minimum (`min_depth`) and maximum (`max_depth`) number of dimensions acceptable, and other `requirements` for the array. The `dtype` argument needs to be a `PyArray_Descr` structure indicating the desired data-type (including required byteorder). The `dtype` argument may be NULL, indicating that any data-type (and byteorder) is acceptable. Unless `FORCECAST` is present in `flags`, this call will generate an error if the data type cannot be safely obtained from the object. If you want to use NULL for the `dtype` and ensure the array is notswapped then use `PyArray_CheckFromAny`. A value of 0 for either of the depth parameters causes the parameter to be ignored. Any of the following array flags can be added (e.g. using |) to get the `requirements` argument. If your code can handle general (e.g. strided, byte-swapped, or unaligned arrays) then `requirements` may be 0. Also, if `op` is not already an array (or does not expose the array interface), then a new array will be created (and filled from `op` using the sequence protocol). The new array will have `NPY_DEFAULT` as its flags member. The `context` argument is passed to the `__array__` method of `op` and is only used if the array is constructed that way. Almost always this parameter is NULL.

`NPY_C_CONTIGUOUS`
Make sure the returned array is C-style contiguous

`NPY_F_CONTIGUOUS`
Make sure the returned array is Fortran-style contiguous.

`NPY_ALIGNED`
Make sure the returned array is aligned on proper boundaries for its data type. An aligned array has the data pointer and every strides factor as a multiple of the alignment factor for the data-type- descriptor.

`NPY_WRITEABLE`
Make sure the returned array can be written to.

`NPY_ENSURECOPY`
Make sure a copy is made of `op`. If this flag is not present, data is not copied if it can be avoided.

`NPY_ENSUREARRAY`
Make sure the result is a base-class ndarray or bigndarray. By default, if `op` is an instance of a subclass of the bigndarray, an instance of that same subclass is returned. If this flag is set, an ndarray object will be returned instead.

`NPY_FORCECAST`
Force a cast to the output type even if it cannot be done safely. Without this flag, a data cast will occur
only if it can be done safely, otherwise an error is raised.

**NPY_UPDATEIFCOPY**

If `op` is already an array, but does not satisfy the requirements, then a copy is made (which will satisfy the requirements). If this flag is present and a copy (of an object that is already an array) must be made, then the corresponding `NPY_UPDATEIFCOPY` flag is set in the returned copy and `op` is made to be read-only. When the returned copy is deleted (presumably after your calculations are complete), its contents will be copied back into `op` and the `op` array will be made writeable again. If `op` is not writeable to begin with, then an error is raised. If `op` is not already an array, then this flag has no effect.

**NPY_BEHAVED**

NPY_ALIGNED | NPY_WRITEABLE

**NPY_CARRAY**

NPY_C_CONTIGUOUS | NPY_BEHAVED

**NPY_CARRAY_RO**

NPY_C_CONTIGUOUS | NPY_ALIGNED

**NPY_FARRAY**

NPY_F_CONTIGUOUS | NPY_BEHAVED

**NPY_FARRAY_RO**

NPY_F_CONTIGUOUS | NPY_ALIGNED

**NPY_DEFAULT**

NPY_CARRAY

**NPY_IN_ARRAY**

NPY_CONTIGUOUS | NPY_ALIGNED

**NPY_IN_FARRAY**

NPY_F_CONTIGUOUS | NPY_ALIGNED

**NPY_INOUT_ARRAY**

NPY_C_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED

**NPY_INOUT_FARRAY**

NPY_F_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED

**NPY_OUT_ARRAY**

NPY_C_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED | NPY_UPDATEIFCOPY

**NPY_OUT_FARRAY**

NPY_F_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED | NPY_UPDATEIFCOPY

PyObject* **PyArray_CheckFromAny** (PyObject* `op`, PyArray_Descr* `dtype`, int `min_depth`, int `max_depth`, int `requirements`, PyObject* `context`)

Nearly identical to **PyArray_FromAny** (...) except `requirements` can contain NPY_NOTSWAPPED (over-riding the specification in `dtype`) and NPY_ELEMENTSTRIDES which indicates that the array should be aligned in the sense that the strides are multiples of the element size.

**NPY_NOTSWAPPED**

Make sure the returned array has a data-type descriptor that is in machine byte-order, over-riding any specification in the `dtype` argument. Normally, the byte-order requirement is determined by the `dtype` argument. If this flag is set and the `dtype` argument does not indicate a machine byte-order descriptor (or is NULL and the object is already an array with a data-type descriptor that is not in machine byte-order), then a new data-type descriptor is created and used with its byte-order field set to native.

**NPY_BEHAVED_NS**

NPY_ALIGNED | NPY_WRITEABLE | NPY_NOTSWAPPED

5.4. Array API
NPY_ELEMENTSTRIDES
Make sure the returned array has strides that are multiples of the element size.

PyObject* PyArray_FromArray (PyArrayObject* op, PyArray_Descr* newtype, int requirements)
Special case of PyArray_FromAny for when op is already an array but it needs to be of a specific newtype
(including byte-order) or has certain requirements.

PyObject* PyArray_FromStructInterface (PyObject* op)
Returns an ndarray object from a Python object that exposes the __array_struct__ method and follows
the array interface protocol. If the object does not contain this method then a borrowed reference to
Py_NotImplemented is returned.

PyObject* PyArray_FromInterface (PyObject* op)
Returns an ndarray object from a Python object that exposes the __array_shape__ and
__array_typestr__ methods following the array interface protocol. If the object does not contain
one of these method then a borrowed reference to Py_NotImplemented is returned.

PyObject* PyArray_FromArrayAttr (PyObject* op, PyArray_Descr* dtype, PyObject* context)
Return an ndarray object from a Python object that exposes the __array__ method. The __array__
method can take 0, 1, or 2 arguments ([dtype, context]) where context is used to pass information about where
the __array__ method is being called from (currently only used in ufuncs).

PyObject* PyArray_ContiguousFromAny (PyObject* op, int typenum, int min_depth, int max_depth)
This function returns a (C-style) contiguous and behaved function array from any nested sequence or array
interface exporting object, op, of (non-flexible) type given by the enumerated typenum, of minimum depth
min_depth, and of maximum depth max_depth. Equivalent to a call to PyArray_FromAny with requirements
set to NPY_DEFAULT and the type_num member of the type argument set to typenum.

PyObject * PyArray_FromObject (PyObject *op, int typenum, int min_depth, int max_depth)
Return an aligned and in native-byteorder array from any nested sequence or array-interface exporting object,
op, of a type given by the enumerated typenum. The minimum number of dimensions the array can have is
given by min_depth while the maximum is max_depth. This is equivalent to a call to PyArray_FromAny
with requirements set to BEHAVED.

 PyObject* PyArray_EnsureArray (PyObject* op)
This function steals a reference to op and makes sure that op is a base-class ndarray. It special cases array
scalars, but otherwise calls PyArray_FromAny (op, NULL, 0, 0, NPY_ENSUREARRAY).

PyObject* PyArray_FromString (char* string, npy_intp slen, PyArray_Descr* dtype, npy_intp num, char* sep)
Construct a one-dimensional ndarray of a single type from a binary or (ASCII) text string of length slen.
The data-type of the array to-be-created is given by dtype. If num is -1, then copy the entire string and return
an appropriately sized array, otherwise, num is the number of items to copy from the string. If sep is NULL
(or ""), then interpret the string as bytes of binary data, otherwise convert the sub-strings separated by sep to
items of data-type dtype. Some data-types may not be readable in text mode and an error will be raised if that
occurs. All errors return NULL.

PyObject* PyArray_FromFile (FILE* fp, PyArray_Descr* dtype, npy_intp num, char* sep)
Construct a one-dimensional ndarray of a single type from a binary or text file. The open file pointer is fp, the
data-type of the array to be created is given by dtype. This must match the data in the file. If num is -1, then
read until the end of the file and return an appropriately sized array, otherwise, num is the number of items to
read. If sep is NULL (or ""), then read from the file in binary mode, otherwise read from the file in text mode
with sep providing the item separator. Some array types cannot be read in text mode in which case an error is
raised.

PyObject* PyArray_FromBuffer (PyObject* buf, PyArray_Descr* dtype, npy_intp count, npy_intp offset)
Construct a one-dimensional ndarray of a single type from an object, buf, that exports the (single-segment)
buffer protocol (or has an attribute __buffer__ that returns an object that exports the buffer protocol). A
writeable buffer will be tried first followed by a read-only buffer. The NPY_WRITEABLE flag of the returned
array will reflect which one was successful. The data is assumed to start at offset bytes from the start of the memory location for the object. The type of the data in the buffer will be interpreted depending on the data-type descriptor, dtype. If count is negative then it will be determined from the size of the buffer and the requested itemsize, otherwise, count represents how many elements should be converted from the buffer.

```c
int PyArray_CopyInto (PyArrayObject* dest, PyArrayObject* src)
```

Copy from the source array, src, into the destination array, dest, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of src must be broadcastable to the shape of dest. The data areas of dest and src must not overlap.

```c
int PyArray_MoveInto (PyArrayObject* dest, PyArrayObject* src)
```

Move data from the source array, src, into the destination array, dest, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of src must be broadcastable to the shape of dest. The data areas of dest and src may overlap.

```c
PyArrayObject* PyArray_GETCONTIGUOUS (PyObject* op)
```

If op is already (C-style) contiguous and well-behaved then just return a reference, otherwise return a (contiguous and well-behaved) copy of the array. The parameter op must be a (sub-class of an) ndarray and no checking for that is done.

```c
PyObject* PyArray_FROM_O (PyObject* obj)
```

Convert obj to an ndarray. The argument can be any nested sequence or object that exports the array interface. This is a macro form of PyArray_FromAny using NULL, 0, 0, 0 for the other arguments. Your code must be able to handle any data-type descriptor and any combination of data-flags to use this macro.

```c
PyObject* PyArray_FROM_OF (PyObject* obj, int requirements)
```

Similar to PyArray_FROM_O except it can take an argument of requirements indicating properties the resulting array must have. Available requirements that can be enforced are NPY_CONTIGUOUS, NPY_F_CONTIGUOUS, NPY_ALIGNED, NPY_WRITEABLE, NPY_NOTSWAPPED, NPY_ENSURECOPY, NPY_UPDATEIFCOPY, NPY_FORCECAST, and NPY_ENSUREARRAY. Standard combinations of flags can also be used:

```c
PyObject* PyArray_FROM_OT (PyObject* obj, int typenum)
```

Similar to PyArray_FROM_O except it can take an argument of typenum specifying the type-number the returned array.

```c
PyObject* PyArray_FROM_OTF (PyObject* obj, int typenum, int requirements)
```

Combination of PyArray_FROM_OF and PyArray_FROM_OT allowing both a typenum and a flags argument to be provided.

```c
PyObject* PyArray_FROMANY (PyObject* obj, int typenum, int min, int max, int requirements)
```

Similar to PyArray_FromAny except the data-type is specified using a typenum. PyArray_DescrFromType (typenum) is passed directly to PyArray_FromAny. This macro also adds NPY_DEFAULT to requirements if NPY_ENSURECOPY is passed in as requirements.

```c
PyObject* PyArray_CheckAxis (PyObject* obj, int* axis, int requirements)
```

Encapsulate the functionality of functions and methods that take the axis= keyword and work properly with None as the axis argument. The input array is obj, while *axis is a converted integer (so that >=MAXDIMS is the None value), and requirements gives the needed properties of obj. The output is a converted version of the input so that requirements are met and if needed a flattening has occurred. On output negative values of *axis are converted and the new value is checked to ensure consistency with the shape of obj.

## 5.4.3 Dealing with types

### General check of Python Type

```c
PyArray_Check (op)
```

Evaluates true if op is a Python object whose type is a sub-type of PyArray_Type.
**PyArray_CheckExact** *(op)*
Evaluates true if *op* is a Python object with type `PyArray_Type`.

**PyArray_HasArrayInterface** *(op, out)*
If *op* implements any part of the array interface, then *out* will contain a new reference to the newly created ndarray using the interface or *out* will contain NULL if an error during conversion occurs. Otherwise, out will contain a borrowed reference to `Py_NotImplemented` and no error condition is set.

**PyArray_HasArrayInterfaceType** *(op, type, context, out)*
If *op* implements any part of the array interface, then *out* will contain a new reference to the newly created ndarray using the interface or *out* will contain NULL if an error during conversion occurs. Otherwise, out will contain a borrowed reference to `Py_NotImplemented` and no error condition is set. This version allows setting of the type and context in the part of the array interface that looks for the `__array__` attribute.

**PyArray_IsZeroDim** *(op)*
Evaluates true if *op* is an instance of (a subclass of) `PyArray_Type` and has 0 dimensions.

**PyArray_IsScalar** *(op, cls)*
Evaluates true if *op* is an instance of `Py{cls}ArrType_Type`.

**PyArray_CheckScalar** *(op)*
Evaluates true if *op* is either an array scalar (an instance of a sub-type of `PyGenericArr_Type`), or an instance of (a sub-class of) `PyArray_Type` whose dimensionality is 0.

**PyArray_IsPythonScalar** *(op)*
Evaluates true if *op* is a builtin Python “scalar” object (int, float, complex, str, unicode, long, bool).

**PyArray_IsAnyScalar** *(op)*
Evaluates true if *op* is either a Python scalar or an array scalar (an instance of a sub-type of `PyGenericArr_Type`).

---

**Data-type checking**

For the typenum macros, the argument is an integer representing an enumerated array data type. For the array type checking macros the argument must be a `PyObject *` that can be directly interpreted as a `PyArrayObject *`.

**PyTypeNum_ISUNSIGNED** *(num)*

**PyDataType_ISUNSIGNED** *(descr)*

**PyArray_ISUNSIGNED** *(obj)*
Type represents an unsigned integer.

**PyTypeNum_ISSIGNED** *(num)*

**PyDataType_ISSIGNED** *(descr)*

**PyArray_ISSIGNED** *(obj)*
Type represents a signed integer.

**PyTypeNum_ISINTEGER** *(num)*

**PyDataType_ISINTEGER** *(descr)*

**PyArray_ISINTEGER** *(obj)*
Type represents a signed integer.
PyArray_ISINTEGER(obj)
Type represents any integer.

PyTypeNum_ISFLOAT(num)

PyDataType_ISFLOAT(descr)

PyArray_ISFLOAT(obj)
Type represents any floating point number.

PyTypeNum_ISCOMPLEX(num)

PyDataType_ISCOMPLEX(descr)

PyArray_ISCOMPLEX(obj)
Type represents any complex floating point number.

PyTypeNum_ISNUMBER(num)

PyDataType_ISNUMBER(descr)

PyArray_ISNUMBER(obj)
Type represents any integer, floating point, or complex floating point number.

PyTypeNum_ISSTRING(num)

PyDataType_ISSTRING(descr)

PyArray_ISSTRING(obj)
Type represents a string data type.

PyTypeNum_ISPYTHON(num)

PyDataType_ISPYTHON(descr)

PyArray_ISPYTHON(obj)
Type represents an enumerated type corresponding to one of the standard Python scalar (bool, int, float, or complex).

PyTypeNum_ISFLEXIBLE(num)

PyDataType_ISFLEXIBLE(descr)

PyArray_ISFLEXIBLE(obj)
Type represents one of the flexible array types (NPY_STRING, NPY_UNICODE, or NPY_VOID).

PyTypeNum_ISUSERDEF(num)

PyDataType_ISUSERDEF(descr)
PyArray_ISUSERDEF(obj)
Type represents a user-defined type.

PyTypeNum_ISEXTEDNUM(num)

PyDataType_ISEXTEDNUM(descr)

PyArray_ISEXTEDNUM(obj)
Type is either flexible or user-defined.

PyTypeNum_ISOBJECT(num)

PyDataType_ISOBJECT(descr)

PyArray_ISOBJECT(obj)
Type represents object data type.

PyTypeNum_ISBOOL(num)

PyDataType_ISBOOL(descr)

PyArray_ISBOOL(obj)
Type represents Boolean data type.

PyDataType_HASFIELDS(descr)

PyArray_HASFIELDS(obj)
Type has fields associated with it.

PyArray_ISNOTSWAPPED(m)
Evaluates true if the data area of the ndarray m is in machine byte-order according to the array’s data-type descriptor.

PyArray_ISBYTESWAPPED(m)
Evaluates true if the data area of the ndarray m is not in machine byte-order according to the array’s data-type descriptor.

Bool PyArray_EquivTypes(PyArray_Descr* type1, PyArray_Descr* type2)
Return NPY_TRUE if type1 and type2 actually represent equivalent types for this platform (the fortran member of each type is ignored). For example, on 32-bit platforms, NPY_LONG and NPY_INT are equivalent. Otherwise return NPY_FALSE.

Bool PyArray_EquivArrTypes(PyArrayObject* a1, PyArrayObject * a2)
Return NPY_TRUE if a1 and a2 are arrays with equivalent types for this platform.

Bool PyArray_EquivTypenums(int typenum1, int typenum2)
Special case of PyArray_EquivTypes(...) that does not accept flexible data types but may be easier to call.

int PyArray_EquivByteorders([byteorder] b1, [byteorder] b2)
True if byteorder characters (NPY_LITTLE, NPY_BIG, NPY_NATIVE, NPY_IGNORE) are either equal or equivalent as to their specification of a native byte order. Thus, on a little-endian machine NPY_LITTLE and NPY_NATIVE are equivalent where they are not equivalent on a big-endian machine.
Converting data types

PyObject* **PyArray_Cast** *(PyArrayObject* `arr`, int `typenum`)*
Mainly for backwards compatibility to the Numeric C-API and for simple casts to non-flexible types. Return a new array object with the elements of `arr` cast to the data-type `typenum` which must be one of the enumerated types and not a flexible type.

PyObject* **PyArray_CastToType** *(PyArrayObject* `arr`, PyArray_Descr* `type`, int `fortran`)*
Return a new array of the `type` specified, casting the elements of `arr` as appropriate. The fortran argument specifies the ordering of the output array.

int **PyArray_CastTo** *(PyArrayObject* `out`, PyArrayObject* `in`)*
Cast the elements of the array `in` into the array `out`. The output array should be writeable, have an integer-multiple of the number of elements in the input array (more than one copy can be placed in out), and have a data type that is one of the builtin types. Returns 0 on success and -1 if an error occurs.

PyObject* VectorUnaryFunc* **PyArray_GetCastFunc** *(PyArray_Descr* `from`, int `totype`)*
Return the low-level casting function to cast from the given descriptor to the builtin type number. If no casting function exists return NULL and set an error. Using this function instead of direct access to `from` -f->cast will allow support of any user-defined casting functions added to a descriptors casting dictionary.

int **PyArray_CanCastSafely** *(int `fromtype`, int `totype`)*
Returns non-zero if an array of data type `fromtype` can be cast to an array of data type `totype` without losing information. An exception is that 64-bit integers are allowed to be cast to 64-bit floating point values even though this can lose precision on large integers so as not to proliferate the use of long doubles without explicit requests. Flexible array types are not checked according to their lengths with this function.

int **PyArray_CanCastTo** *(PyArray_Descr* `fromtype`, PyArray_Descr* `totype`)*
Returns non-zero if an array of data type `fromtype` (which can include flexible types) can be cast safely to an array of data type `totype` (which can include flexible types). This is basically a wrapper around `PyArray_CanCastSafely` with additional support for size checking if `fromtype` and `totype` are NPY_STRING or NPY_UNICODE.

int **PyArray_ObjectType** *(PyObject* `op`, int `mintype`)*
This function is useful for determining a common type that two or more arrays can be converted to. It only works for non-flexible array types as no itemsize information is passed. The `mintype` argument represents the minimum type acceptable, and `op` represents the object that will be converted to an array. The return value is the enumerated typenumber that represents the data-type that `op` should have.

void **PyArray_ArrayType** *(PyObject* `op`, PyArray_Descr* `mintype`, PyArray_Descr* `outtype`)*
This function works similarly to `PyArray_ObjectType (...)` except it handles flexible arrays. The `mintype` argument can have an itemsize member and the `outtype` argument will have an itemsize member at least as big but perhaps bigger depending on the object `op`.

PyArrayObject** **PyArray_ConvertToCommonType** *(PyObject* `obj`, int* `n`)*
Convert a sequence of Python objects contained in `obj` to an array of ndarrays each having the same data type. The type is selected based on the typenumber (larger type number is chosen over a smaller one) ignoring objects that are only scalars. The length of the sequence is returned in `n`, and an `n` -length array of `PyArrayObject` pointers is the return value (or NULL if an error occurs). The returned array must be freed by the caller of this routine (using `PyDataMem_FREE` ) and all the array objects in it DECREF ‘d or a memory-leak will occur. The example template-code below shows a typically usage:

```c
mps = PyArray_ConvertToCommonType(obj, &n);
if (mps==NULL) return NULL;
{code}
<before return>
for (i=0; i<n; i++) Py_DECREF(mps[i]);
```
char* PyArray_Zero (PyArrayObject* arr)
A pointer to newly created memory of size arr->itemsize that holds the representation of 0 for that type. The returned pointer, ret, must be freed using PyDataMem_FREE (ret) when it is not needed anymore.

char* PyArray_One (PyArrayObject* arr)
A pointer to newly created memory of size arr->itemsize that holds the representation of 1 for that type. The returned pointer, ret, must be freed using PyDataMem_FREE (ret) when it is not needed anymore.

int PyArray_ValidType (int typenum)
Returns NPY_TRUE if typenum represents a valid type-number (builtin or user-defined or character code). Otherwise, this function returns NPY_FALSE.

New data types

void PyArray_InitArrFuncs (PyArray_ArrFuncs*f)
Initialize all function pointers and members to NULL.

int PyArray_RegisterDataType (PyArray_Descr* dtype)
Register a data-type as a new user-defined data type for arrays. The type must have most of its entries filled in. This is not always checked and errors can produce segfaults. In particular, the typeobj member of the dtype structure must be filled with a Python type that has a fixed-size element-size that corresponds to the elsize member of dtype. Also the f member must have the required functions: nonzero, copyswap, copyswapn, getitem, setitem, and cast (some of the cast functions may be NULL if no support is desired). To avoid confusion, you should choose a unique character type code but this is not enforced and not relied on internally.

A user-defined type number is returned that uniquely identifies the type. A pointer to the new structure can then be obtained from PyArray_DescrFromType using the returned type number. A -1 is returned if an error occurs. If this dtype has already been registered (checked only by the address of the pointer), then return the previously-assigned type-number.

int PyArray_RegisterCastFunc (PyArray_Descr* descr, int totype, PyArray_VectorUnaryFunc* castfunc)
Register a low-level casting function, castfunc, to convert from the data-type, descr, to the given data-type number, totype. Any old casting function is over-written. A 0 is returned on success or a -1 on failure.

int PyArray_RegisterCanCast (PyArray_Descr* descr, int totype, PyArray_SCALARKIND scalar)
Register the data-type number, totype, as castable from data-type object, descr, of the given scalar kind. Use scalar = NPY_NOSCALAR to register that an array of data-type descr can be cast safely to a data-type whose type_number is totype.

Special functions for PyArray_OBJECT

int PyArray_INCREF (PyArrayObject* op)
Used for an array, op, that contains any Python objects. It increments the reference count of every object in the array according to the data-type of op. A -1 is returned if an error occurs, otherwise 0 is returned.

void PyArray_Item_INCREF (char* ptr, PyArray_Descr* dtype)
A function to INCREASE all the objects at the location ptr according to the data-type dtype. If ptr is the start of a record with an object at any offset, then this will (recursively) increment the reference count of all object-like items in the record.

int PyArray_XDECREF (PyArrayObject* op)
Used for an array, op, that contains any Python objects. It decrements the reference count of every object in the array according to the data-type of op. Normal return value is 0. A -1 is returned if an error occurs.
void PyArray_Item_XDECREF (char* ptr, PyArray_Descr* dtype)
A function to XDECREF all the object-like items at the location ptr as recorded in the data-type, dtype.
This works recursively so that if dtype itself has fields with data-types that contain object-like items, all the
object-like fields will be XDECREF’d.

void PyArray_FillObjectArray (PyArrayObject* arr, PyObject* obj)
Fill a newly created array with a single value obj at all locations in the structure with object data-types. No
checking is performed but arr must be of data-type PyArray_OBJECT and be single-segment and uninitialized
(no previous objects in position). Use PyArray_DECREF (arr) if you need to decrement all the items in
the object array prior to calling this function.

5.4.4 Array flags

The flags attribute of the PyArrayObject structure contains important information about the memory used by
the array (pointed to by the data member) This flag information must be kept accurate or strange results and even
segfaults may result.

There are 6 (binary) flags that describe the memory area used by the data buffer. These constants are defined in
arrayobject.h and determine the bit-position of the flag. Python exposes a nice attribute-based interface as well
as a dictionary-like interface for getting (and, if appropriate, setting) these flags.

Memory areas of all kinds can be pointed to by an ndarray, necessitating these flags. If you get an arbitrary
PyArrayObject in C-code, you need to be aware of the flags that are set. If you need to guarantee a certain kind
of array (like NPY_CONTIGUOUS and NPY_BEHAVED), then pass these requirements into the PyArray_FromAny
function.

Basic Array Flags

An ndarray can have a data segment that is not a simple contiguous chunk of well-behaved memory you can manip-
ulate. It may not be aligned with word boundaries (very important on some platforms). It might have its data in a
different byte-order than the machine recognizes. It might not be writeable. It might be in Fortran-contiguous order.
The array flags are used to indicate what can be said about data associated with an array.

NPY_C_CONTIGUOUS
The data area is in C-style contiguous order (last index varies the fastest).

NPY_F_CONTIGUOUS
The data area is in Fortran-style contiguous order (first index varies the fastest).

Notice that contiguous 1-d arrays are always both NPY_FORTRAN contiguous and C contiguous. Both of these flags
can be checked and are convenience flags only as whether or not an array is NPY_CONTIGUOUS or NPY_FORTRAN
can be determined by the strides, dimensions, and itemsize attributes.

NPY_OWNDATA
The data area is owned by this array.

NPY_ALIGNED
The data area is aligned appropriately (for all strides).

NPY_WRITEABLE
The data area can be written to.

Notice that the above 3 flags are are defined so that a new, well-behaved array has these flags defined as true.

NPY_UPDATEIFCOPY
The data area represents a (well-behaved) copy whose information should be transferred back to the original
when this array is deleted.
This is a special flag that is set if this array represents a copy made because a user required certain flags in `PyArray_FromAny` and a copy had to be made of some other array (and the user asked for this flag to be set in such a situation). The base attribute then points to the “misbehaved” array (which is set read_only). When the array with this flag set is deallocated, it will copy its contents back to the “misbehaved” array (casting if necessary) and will reset the “misbehaved” array to `NPY_WRITEABLE`. If the “misbehaved” array was not `NPY_WRITEABLE` to begin with then `PyArray_FromAny` would have returned an error because `NPY_UPDATEIFCOPY` would not have been possible.

`PyArray_UpdateFlags` (obj, flags) will update the obj->flags for flags which can be any of `NPY_CONTIGUOUS`, `NPY_FORTRAN`, `NPY_ALIGNED`, or `NPY_WRITEABLE`.

**Combinations of array flags**

- `NPY_BEHAVED | NPY_ALIGNED | NPY_WRITEABLE`
- `NPY_CARRAY | NPY_C_CONTIGUOUS | NPY_BEHAVED`
- `NPY_CARRAY_RO | NPY_C_CONTIGUOUS | NPY_ALIGNED`
- `NPY_FARRAY | NPY_F_CONTIGUOUS | NPY_BEHAVED`
- `NPY_FARRAY_RO | NPY_F_CONTIGUOUS | NPY_ALIGNED`
- `NPY_DEFAULT | NPY_CARRAY`
- `NPY_UPDATE_ALL | NPY_C_CONTIGUOUS | NPY_F_CONTIGUOUS | NPY_ALIGNED`

**Flag-like constants**

These constants are used in `PyArray_FromAny` (and its macro forms) to specify desired properties of the new array.

- `NPY_FORCECAST`
  Cast to the desired type, even if it can’t be done without losing information.

- `NPY_ENSURECOPY`
  Make sure the resulting array is a copy of the original.

- `NPY_ENSUREARRAY`
  Make sure the resulting object is an actual ndarray (or bigndarray), and not a sub-class.

- `NPY_NOTSWAPPED`
  Only used in `PyArray_CheckFromAny` to over-ride the byteorder of the data-type object passed in.

- `NPY_BEHAVED_NS | NPY_ALIGNED | NPY_WRITEABLE | NPY_NOTSWAPPED`

**Flag checking**

For all of these macros arr must be an instance of a (subclass of) `PyArray_Type`, but no checking is done.
**PyArray_CHKFLAGS** (arr, flags)
The first parameter, arr, must be an ndarray or subclass. The parameter, flags, should be an integer consisting of bitwise combinations of the possible flags an array can have: NPY_C_CONTIGUOUS, NPY_F_CONTIGUOUS, NPY_OWNDATA, NPY_ALIGNED, NPY_WRITEABLE, NPY_UPDATEIFCOPY.

**PyArray_ISCONTIGUOUS** (arr)
Evaluates true if arr is C-style contiguous.

**PyArray_ISFORTRAN** (arr)
Evaluates true if arr is Fortran-style contiguous.

**PyArray_ISWRITEABLE** (arr)
Evaluates true if the data area of arr can be written to.

**PyArray_ISALIGNED** (arr)
Evaluates true if the data area of arr is properly aligned on the machine.

**PyArray_ISBEHAVED** (arr)
Evaluates true if the data area of arr is aligned and writeable and in machine byte-order according to its descriptor.

**PyArray_ISBEHAVED_RO** (arr)
Evaluates true if the data area of arr is aligned and in machine byte-order.

**PyArray_ISCARRAY** (arr)
Evaluates true if the data area of arr is C-style contiguous, and PyArray_ISBEHAVED (arr) is true.

**PyArray_ISFARRAY** (arr)
Evaluates true if the data area of arr is Fortran-style contiguous and PyArray_ISBEHAVED (arr) is true.

**PyArray_ISCARRAY_RO** (arr)
Evaluates true if the data area of arr is C-style contiguous, aligned, and in machine byte-order.

**PyArray_ISFARRAY_RO** (arr)
Evaluates true if the data area of arr is Fortran-style contiguous, aligned, and in machine byte-order.

**PyArray_ISONESEGMENT** (arr)
Evaluates true if the data area of arr consists of a single (C-style or Fortran-style) contiguous segment.

void **PyArray_UpdateFlags** (PyArrayObject* arr, int flagmask)
The NPY_C_CONTIGUOUS, NPY_ALIGNED, and NPY_F_CONTIGUOUS array flags can be “calculated” from the array object itself. This routine updates one or more of these flags of arr as specified in flagmask by performing the required calculation.

**Warning:** It is important to keep the flags updated (using PyArray_UpdateFlags can help) whenever a manipulation with an array is performed that might cause them to change. Later calculations in NumPy that rely on the state of these flags do not repeat the calculation to update them.

### 5.4.5 Array method alternative API

#### Conversion

PyObject* **PyArray_GetField** (PyArrayObject* self, PyArray_Descr* dtype, int offset)
Equivalent to ndarray.getfield (self, dtype, offset). Return a new array of the given dtype using the data in the current array at a specified offset in bytes. The offset plus the itemsize of the new array type must be less than self->descr->elsize or an error is raised. The same shape and strides as the original array are used. Therefore, this function has the effect of returning a field from a record array. But, it can also be used to select specific bytes or groups of bytes from any array type.
int PyArray_SetField (PyArrayObject* self, PyArray_Descr* dtype, int offset, PyObject* val)
   Equivalent to ndarray.setfield (self, val, dtype, offset). Set the field starting at offset in bytes and of the given dtype to val. The offset plus dtype->elsize must be less than self->descr->elsize or an error is raised. Otherwise, the val argument is converted to an array and copied into the field pointed to. If necessary, the elements of val are repeated to fill the destination array. But, the number of elements in the destination must be an integer multiple of the number of elements in val.

PyObject* PyArray_Byteswap (PyArrayObject* self, Bool inplace)
   Equivalent to ndarray.byteswap (self, inplace). Return an array whose data area is byteswapped. If inplace is non-zero, then do the byteswap inplace and return a reference to self. Otherwise, create a byteswapped copy and leave self unchanged.

PyObject* PyArray_NewCopy (PyArrayObject* old, NPY_ORDER order)
   Equivalent to ndarray.copy (self, fortran). Make a copy of the old array. The returned array is always aligned and writeable with data interpreted the same as the old array. If order is NPY_CORDER, then a C-style contiguous array is returned. If order is NPY_FORTRANORDER, then a Fortran-style contiguous array is returned. If order is NPY_ANYORDER, then the array returned is Fortran-style contiguous only if the old one is; otherwise, it is C-style contiguous.

PyObject* PyArray_ToList (PyArrayObject* self)
   Equivalent to ndarray.tolist (self). Return a nested Python list from self.

PyObject* PyArray_ToString (PyArrayObject* self, NPY_ORDER order)
   Equivalent to ndarray.tostring (self, order). Return the bytes of this array in a Python string.

PyObject* PyArray_ToFile (PyArrayObject* self, FILE* fp, char* sep, char* format)
   Write the contents of self to the file pointer fp in C-style contiguous fashion. Write the data as binary bytes if sep is the string ** or NULL. Otherwise, write the contents of self as text using the sep string as the item separator. Each item will be printed to the file. If the format string is not NULL or **, then it is a Python print statement format string showing how the items are to be written.

int PyArray_Dump (PyObject* self, PyObject* file, int protocol)
   Pickle the object in self to the given file (either a string or a Python file object). If file is a Python string it is considered to be the name of a file which is then opened in binary mode. The given protocol is used (if protocol is negative, or the highest available is used). This is a simple wrapper around cPickle.dump (self, file, protocol).

PyObject* PyArray_Dumps (PyObject* self, int protocol)
   Pickle the object in self to a Python string and return it. Use the Pickle protocol provided (or the highest available if protocol is negative).

int PyArray_FillWithScalar (PyArrayObject* arr, PyObject* obj)
   Fill the array, arr, with the given scalar object, obj. The object is first converted to the data type of arr, and then copied into every location. A -1 is returned if an error occurs, otherwise 0 is returned.

PyObject* PyArray_View (PyArrayObject* self, PyArray_Descr* dtype)
   Equivalent to ndarray.view (self, dtype). Return a new view of the array self as possibly a different data-type, dtype. If dtype is NULL, then the returned array will have the same data type as self. The new data-type must be consistent with the size of self. Either the itemizes must be identical, or self must be single-segment and the total number of bytes must be the same. In the latter case the dimensions of the returned array will be altered in the last (or first for Fortran-style contiguous arrays) dimension. The data area of the returned array and self is exactly the same.

Shape Manipulation

PyObject* PyArray_Newshape (PyArrayObject* self, PyArray_Dims* newshape)
   Result will be a new array (pointing to the same memory location as self if possible), but having a shape given by newshape. If the new shape is not compatible with the strides of self, then a copy of the array with the new specified shape will be returned.
PyObject* **PyArray_Reshape** (PyArrayObject* self, PyObject* shape)
Equivalent to ndarray.reshape (self, shape) where shape is a sequence. Converts shape to a PyArray_Dims structure and calls PyArray_Newshape internally.

PyObject* **PyArray_Squeeze** (PyArrayObject* self)
Equivalent to ndarray.squeeze (self). Return a new view of self with all of the dimensions of length 1 removed from the shape.

**Warning:** matrix objects are always 2-dimensional. Therefore, **PyArray_Squeeze** has no effect on arrays of matrix sub-class.

PyObject* **PyArray_SwapAxes** (PyArrayObject* self, int a1, int a2)
Equivalent to ndarray.swapaxes (self, a1, a2). The returned array is a new view of the data in self with the given axes, a1 and a2, swapped.

PyObject* **PyArray_Resize** (PyArrayObject* self, PyArray_Dims* newshape, int refcheck, NPY_ORDER fortran)
Equivalent to ndarray.resize (self, newshape, refcheck = refcheck, order= fortran ). This function only works on single-segment arrays. It changes the shape of self in-place and will reallocate the memory for self if newshape has a different total number of elements then the old shape. If reallocation is necessary, then self must own its data, have self - >base==NULL, have self - >weakrefs==NULL, and (unless refcheck is 0) not be referenced by any other array. A reference to the new array is returned. The fortran argument can be NPY_ANYORDER, NPY_CORDER, or NPY_FORTRANORDER. It currently has no effect. Eventually it could be used to determine how the resize operation should view the data when constructing a differently-dimensioned array.

PyObject* **PyArray_Transpose** (PyArrayObject* self, PyArray_Dims* permute)
Equivalent to ndarray.transpose (self, permute). Permute the axes of the ndarray object self according to the data structure permute and return the result. If permute is NULL, then the resulting array has its axes reversed. For example if self has shape 10 × 20 × 30, and permute .ptr is (0,2,1) the shape of the result is 10 × 30 × 20. If permute is NULL, the shape of the result is 30 × 20 × 10.

PyObject* **PyArray_Flatten** (PyArrayObject* self, NPY_ORDER order)
Equivalent to ndarray.flatten (self, order). Return a 1-d copy of the array. If order is NPY_FORTRANORDER the elements are scanned out in Fortran order (first-dimension varies the fastest). If order is NPY_CORDER, the elements of self are scanned in C-order (last dimension varies the fastest). If order NPY_ANYORDER, then the result of PyArray_ISFORTRAN (self) is used to determine which order to flatten.

PyObject* **PyArray_Ravel** (PyArrayObject* self, NPY_ORDER order)
Equivalent to self.ravel(order). Same basic functionality as PyArray_Flatten (self, order) except if order is 0 and self is C-style contiguous, the shape is altered but no copy is performed.

**Item selection and manipulation**

PyObject* **PyArray_TakeFrom** (PyArrayObject* self, PyObject* indices, int axis, PyArrayObject* ret, NPY_CLIPMODE clipmode)
Equivalent to ndarray.take (self, indices, axis, ret, clipmode) except axis =None in Python is obtained by setting axis = NPY_MAXDIMS in C. Extract the items from self indicated by the integer-valued indices along the given axis. The clipmode argument can be NPY_RAISE, NPY_WRAP, or NPY_CLIP to indicate what to do with out-of-bound indices. The ret argument can specify an output array rather than having one created internally.

PyObject* **PyArray_PutTo** (PyArrayObject* self, PyObject* values, PyObject* indices, NPY_CLIPMODE clipmode)
Equivalent to self.put(values, indices, clipmode ). Put values into self at the corresponding (flattened) indices. If values is too small it will be repeated as necessary.
PyObject* **PyArray_PutMask** *(PyArrayObject* `self`, PyObject* `values`, PyObject* `mask`)*
Place the `values` in `self` wherever corresponding positions (using a flattened context) in `mask` are true. The `mask` and `self` arrays must have the same total number of elements. If `values` is too small, it will be repeated as necessary.

PyObject* **PyArray_Repeat** *(PyArrayObject* `self`, PyObject* `op`, int `axis`)*
Equivalent to `ndarray.repeat` *(`self`, `op`, `axis`)*. Copy the elements of `self`, `op` times along the given `axis`. Either `op` is a scalar integer or a sequence of length `self` ->`dimensions`[ `axis` ] indicating how many times to repeat each item along the `axis`.

PyObject* **PyArray_Choose** *(PyArrayObject* `self`, PyObject* `op`, PyArrayObject* `ret`, NPY_CLIPMODE `clipmode`)*
Equivalent to `ndarray.choose` *(`self`, `op`, `ret`, `clipmode`)*. Create a new array by selecting elements from the sequence of arrays in `op` based on the integer values in `self`. The arrays must all be broadcastable to the same shape and the entries in `self` should be between 0 and `len(op)`. The output is placed in `ret` unless it is NULL in which case a new output is created. The `clipmode` argument determines behavior for when entries in `self` are not between 0 and `len(op)`.

NPY_RAISE
raise a `ValueError`;

NPY_WRAP
wrap values < 0 by adding `len(op)` and values >=`len(op)` by subtracting `len(op)` until they are in range;

NPY_CLIP
all values are clipped to the region [0, `len(op)`).

PyObject* **PyArray_Sort** *(PyArrayObject* `self`, int `axis`)*
Equivalent to `ndarray.sort` *(`self`, `axis`)*. Return an array with the items of `self` sorted along `axis`.

PyObject* **PyArray_ArgSort** *(PyArrayObject* `self`, int `axis`)*
Equivalent to `ndarray.argsort` *(`self`, `axis`)*. Return an array of indices such that selection of these indices along the given `axis` would return a sorted version of `self`. If `self` ->`descr` is a data-type with fields defined, then `self` ->`descr`->`names` is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a record array, create a new data-type with a different order of names and construct a view of the array with that new data-type.

PyObject* **PyArray_LexSort** *(PyObject* `sort_keys`, int `axis`)*
Given a sequence of arrays *(`sort_keys`)* of the same shape, return an array of indices (similar to `PyArray_ArgSort` ( ...)) that would sort the arrays lexicographically. A lexicographic sort specifies that when two keys are found to be equal, the order is based on comparison of subsequent keys. A merge sort (which leaves equal entries unmoved) is required to be defined for the types. The sort is accomplished by sorting the indices first using the first `sort_key` and then using the second `sort_key` and so forth. This is equivalent to the lexsort(*`sort_keys`, `axis`*) Python command. Because of the way the merge-sort works, be sure to understand the order the `sort_keys` must be in (reversed from the order you would use when comparing two elements).

If these arrays are all collected in a record array, then `PyArray_Sort` ( ... ) can also be used to sort the array directly.

PyObject* **PyArray_SearchSorted** *(PyArrayObject* `self`, PyObject* `values`)*
Equivalent to `ndarray.searchsorted` *(`self`, `values`)*. Assuming `self` is a 1-d array in ascending order representing bin boundaries then the output is an array the same shape as `values` of bin numbers, giving the bin into which each item in `values` would be placed. No checking is done on whether or not `self` is in ascending order.

PyObject* **PyArray_Diagonal** *(PyArrayObject* `self`, int `offset`, int `axis1`, int `axis2`)*
Equivalent to `ndarray.diagonal` *(`self`, `offset`, `axis1`, `axis2`)*. Return the `offset` diagonals of the 2-d arrays defined by `axis1` and `axis2`.
PyObject* PyArray_Nonzero (PyArrayObject* self)
Equivalent to ndarray.nonzero (self). Returns a tuple of index arrays that select elements of self that are nonzero. If (nd= PyArray_NDIM (self))==1, then a single index array is returned. The index arrays have data type NPY_INT. If a tuple is returned (nd != 1), then its length is nd.

PyObject* PyArray_Compress (PyArrayObject* self, PyObject* condition, int axis, PyArrayObject* out)
Equivalent to ndarray.compress (self, condition, axis ). Return the elements along axis corresponding to elements of condition that are true.

**Calculation**

**Tip:** Pass in NPY_MAXDIMS for axis in order to achieve the same effect that is obtained by passing in axis = None in Python (treating the array as a 1-d array).

PyObject* PyArray_ArgMax (PyArrayObject* self, int axis)
Equivalent to ndarray.argmax (self, axis). Return the index of the largest element of self along axis.

PyObject* PyArray_ArgMin (PyArrayObject* self, int axis)
Equivalent to ndarray.argmin (self, axis). Return the index of the smallest element of self along axis.

PyObject* PyArray_Max (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.max (self, axis). Return the largest element of self along the given axis.

PyObject* PyArray_Min (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.min (self, axis). Return the smallest element of self along the given axis.

PyObject* PyArray_Ptp (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.ptp (self, axis). Return the difference between the largest element of self along axis and the smallest element of self along axis.

**Note:** The rtype argument specifies the data-type the reduction should take place over. This is important if the data-type of the array is not “large” enough to handle the output. By default, all integer data-types are made at least as large as NPY_LONG for the “add” and “multiply” ufuncs (which form the basis for mean, sum, sumsum, prod, and cunprod functions).

PyObject* PyArray_Mean (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.mean (self, axis, rtype). Returns the mean of the elements along the given axis, using the enumerated type rtype as the data type to sum in. Default sum behavior is obtained using PyArray_NOTYPE for rtype.

PyObject* PyArray_Trace (PyArrayObject* self, int offset, int axis1, int axis2, int rtype, PyArrayObject* out)
Equivalent to ndarray.trace (self, offset, axis1, axis2, rtype). Return the sum (using rtype as the data type of summation) over the offset diagonal elements of the 2-d arrays defined by axis1 and axis2 variables. A positive offset chooses diagonals above the main diagonal. A negative offset selects diagonals below the main diagonal.

PyObject* PyArray_Clipped (PyArrayObject* self, PyObject* min, PyObject* max)
Equivalent to ndarray.clip (self, min, max). Clip an array, self, so that values larger than max are fixed to max and values less than min are fixed to min.

PyObject* PyArray_Conjugate (PyArrayObject* self)
Equivalent to ndarray.conjugate (self). Return the complex conjugate of self. If self is not of complex data type, then return self with an reference.

PyObject* PyArray_Round (PyArrayObject* self, int decimals, PyArrayObject* out)
Equivalent to ndarray.round (self, decimals, out). Returns the array with elements rounded to the nearest decimal place. The decimal place is defined as the 10^-decimals digit so that negative decimals cause rounding to the nearest 10's, 100's, etc. If out is NULL, then the output array is created, otherwise the output is placed in out which must be the correct size and type.
NumPy Reference, Release 1.4.1

PyObject* **PyArray_Sum** (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.sum (self, axis, rtype). Return 1-d vector sums of elements in self along axis. Perform the sum after converting data to data type rtype.

PyObject* **PyArray_CumSum** (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.cumsum (self, axis, rtype). Return cumulative 1-d sums of elements in self along axis. Perform the sum after converting data to data type rtype.

PyObject* **PyArray_Prod** (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.prod (self, axis, rtype). Return 1-d products of elements in self along axis. Perform the product after converting data to data type rtype.

PyObject* **PyArray_CumProd** (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.cumprod (self, axis, rtype). Return 1-d cumulative products of elements in self along axis. Perform the product after converting data to data type rtype.

PyObject* **PyArray_All** (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.all (self, axis). Return an array with True elements for every 1-d sub-array of self defined by axis in which all the elements are True.

PyObject* **PyArray_Any** (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.any (self, axis). Return an array with True elements for every 1-d sub-array of self defined by axis in which any of the elements are True.

5.4.6 Functions

Array Functions

int **PyArray_AsCArray** (PyObject** op, void* ptr, npy_intp* dims, int nd, int typenum, int itemsize)
Sometimes it is useful to access a multidimensional array as a C-style multi-dimensional array so that algorithms can be implemented using C’s a[i][j][k] syntax. This routine returns a pointer, ptr, that simulates this kind of C-style array, for 1-, 2-, and 3-d ndarrays.

Parameters

- **op** – The address to any Python object. This Python object will be replaced with an equivalent well-behaved, C-style contiguous, ndarray of the given data type specific by the last two arguments. Be sure that stealing a reference in this way to the input object is justified.
- **ptr** – The address to a (ctype* for 1-d, ctype** for 2-d or ctype*** for 3-d) variable where ctype is the equivalent C-type for the data type. On return, ptr will be addressable as a 1-d, 2-d, or 3-d array.
- **dims** – An output array that contains the shape of the array object. This array gives boundaries on any looping that will take place.
- **nd** – The dimensionality of the array (1, 2, or 3).
- **typenum** – The expected data type of the array.
- **itemsize** – This argument is only needed when typenum represents a flexible array. Otherwise it should be 0.
Note: The simulation of a C-style array is not complete for 2-d and 3-d arrays. For example, the simulated arrays of pointers cannot be passed to subroutines expecting specific, statically-defined 2-d and 3-d arrays. To pass to functions requiring those kind of inputs, you must statically define the required array and copy data.

```c
int PyArray_Free(PyObject* op, void* ptr)
```
Must be called with the same objects and memory locations returned from `PyArray_AsCArray(...)`. This function cleans up memory that otherwise would get leaked.

```c
PyObject* PyArray_Concatenate(PyObject* obj, int axis)
```
Join the sequence of objects in `obj` together along `axis` into a single array. If the dimensions or types are not compatible an error is raised.

```c
PyObject* PyArray_InnerProduct(PyObject* obj1, PyObject* obj2)
```
Compute a product-sum over the last dimensions of `obj1` and `obj2`. Neither array is conjugated.

```c
PyObject* PyArray_MatrixProduct(PyObject* obj1, PyObject* obj)
```
Compute a product-sum over the last dimension of `obj1` and the second-to-last dimension of `obj2`. For 2-d arrays this is a matrix-product. Neither array is conjugated.

```c
PyObject* PyArray_CopyAndTranspose(PyObject* op)
```
A specialized copy and transpose function that works only for 2-d arrays. The returned array is a transposed copy of `op`.

```c
PyObject* PyArray_Correlate(PyObject* op1, PyObject* op2, int mode)
```
Compute the 1-d correlation of the 1-d arrays `op1` and `op2`. The correlation is computed at each output point by multiplying `op1` by a shifted version of `op2` and summing the result. As a result of the shift, needed values outside of the defined range of `op1` and `op2` are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as `op1`, 2 - return all possible shifts (any overlap at all is accepted).

**Notes**
This does not compute the usual correlation: if `op2` is larger than `op1`, the arguments are swapped, and the conjugate is never taken for complex arrays. See `PyArray_Correlate2` for the usual signal processing correlation.

```c
PyObject* PyArray_Correlate2(PyObject* op1, PyObject* op2, int mode)
```
Updated version of `PyArray_Correlate`, which uses the usual definition of correlation for 1d arrays. The correlation is computed at each output point by multiplying `op1` by a shifted version of `op2` and summing the result. As a result of the shift, needed values outside of the defined range of `op1` and `op2` are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as `op1`, 2 - return all possible shifts (any overlap at all is accepted).

**Notes**
Compute `z` as follows:

\[
z[k] = \sum_n \text{op1}[n] \times \text{conj(op2}[n+k])
\]

```c
PyObject* PyArray_Where(PyObject* condition, PyObject* x, PyObject* y)
```
If both `x` and `y` are NULL, then return `PyArray_Nonzero(condition)`. Otherwise, both `x` and `y` must be given and the object returned is shaped like `condition` and has elements of `x` and `y` where `condition` is respectively True or False.

**Other functions**

```c
Bool PyArray_CheckStrides(int elsize, int nd, npy_intp numbytes, npy_intp* dims, npy_intp* newstrides)
```
Determine if `newstrides` is a strides array consistent with the memory of an `nd`-dimensional array with shape `dims` and element-size, `elsize`. The `newstrides` array is checked to see if jumping by the provided number of
bytes in each direction will ever mean jumping more than numbytes which is the assumed size of the available memory segment. If numbytes is 0, then an equivalent numbytes is computed assuming nd, dims, and elsize refer to a single-segment array. Return NPY_TRUE if newstrides is acceptable, otherwise return NPY_FALSE.

npy_intp PyArray_MultiplyList (npy_intp* seq, int n)

int PyArray_MultiplyIntList (int* seq, int n)

Both of these routines multiply an n-length array, seq, of integers and return the result. No overflow checking is performed.

int PyArray_CompareLists (npy_intp* l1, npy_intp* l2, int n)

Given two n-length arrays of integers, l1, and l2, return 1 if the lists are identical; otherwise, return 0.

5.4.7 Array Iterators

An array iterator is a simple way to access the elements of an N-dimensional array quickly and efficiently. Section 2 provides more description and examples of this useful approach to looping over an array.

PyObject* PyArray_IterNew (PyObject* arr)

Return an array iterator object from the array, arr. This is equivalent to arr. flat. The array iterator object makes it easy to loop over an N-dimensional non-contiguous array in C-style contiguous fashion.

PyObject* PyArray_IterAllButAxis (PyObject* arr, int *axis)

Return an array iterator that will iterate over all axes but the one provided in *axis. The returned iterator cannot be used with PyArray_ITER_GOTO1D. This iterator could be used to write something similar to what ufuncs do wherein the loop over the largest axis is done by a separate sub-routine. If *axis is negative then *axis will be set to the axis having the smallest stride and that axis will be used.

PyObject * PyArray_BroadcastToShape (PyObject* arr, npy_intp *dimensions, int nd)

Return an array iterator that is broadcast to iterate as an array of the shape provided by dimensions and nd.

int PyArrayIter_Check (PyObject* op)

Evaluates true if op is an array iterator (or instance of a subclass of the array iterator type).

void PyArray_ITER_RESET (PyObject* iterator)

Reset an iterator to the beginning of the array.

void PyArray_ITER_NEXT (PyObject* iterator)

Increment the index and the dataptr members of the iterator to point to the next element of the array. If the array is not (C-style) contiguous, also increment the N-dimensional coordinates array.

void * PyArray_ITER_DATA (PyObject* iterator)

A pointer to the current element of the array.

void PyArray_ITER_GOTO (PyObject* iterator, npy_intp* destination)

Set the iterator index, dataptr, and coordinates members to the location in the array indicated by the N-dimensional c-array, destination, which must have size at least iterator ->nd_m1+1.

PyArray_ITER_GOTO1D (PyObject* iterator, npy_intp index)

Set the iterator index and dataptr to the location in the array indicated by the integer index which points to an element in the C-styled flattened array.

int PyArray_ITER_NOTDONE (PyObject* iterator)

Evaluates TRUE as long as the iterator has not looped through all of the elements, otherwise it evaluates FALSE.
5.4.8 Broadcasting (multi-iterations)

PyObject* PyArray_MultiIterNew (int num, ...)

A simplified interface to broadcasting. This function takes the number of arrays to broadcast and then num extra (PyObject *) arguments. These arguments are converted to arrays and iterators are created. PyArray_Broadcast is then called on the resulting multi-iterator object. The resulting, broadcasted multi-iterator object is then returned. A broadcasted operation can then be performed using a single loop and using PyArray_MultiIter_NEXT( ...)

void PyArray_MultiIter_RESET (PyObject* multi)

Reset all the iterators to the beginning in a multi-iterator object, multi.

void PyArray_MultiIter_NEXT (PyObject* multi)

Advance each iterator in a multi-iterator object, multi, to its next (broadcasted) element.

void* PyArray_MultiIter_DATA (PyObject* multi, int i)

Return the data-pointer of the i-th iterator in a multi-iterator object.

void PyArray_MultiIter_NEXTi (PyObject* multi, int i)

Advance the pointer of only the i-th iterator.

void PyArray_MultiIter_GOTO (PyObject* multi, npy_intp* destination)

Advance each iterator in a multi-iterator object, multi, to the given N-dimensional destination where N is the number of dimensions in the broadcasted array.

void PyArray_MultiIter_GOTO1D (PyObject* multi, npy_intp index)

Advance each iterator in a multi-iterator object, multi, to the corresponding location of the index into the flattened broadcasted array.

int PyArray_MultiIter_NOTDONE (PyObject* multi)

Evaluates TRUE as long as the multi-iterator has not looped through all of the elements (of the broadcasted result), otherwise it evaluates FALSE.

int PyArray_Broadcast (PyArrayMultiIterObject* mit)

This function encapsulates the broadcasting rules. The mit container should already contain iterators for all the arrays that need to be broadcast. On return, these iterators will be adjusted so that iteration over each simultaneously will accomplish the broadcasting. A negative number is returned if an error occurs.

int PyArray_RemoveSmallest (PyArrayMultiIterObject* mit)

This function takes a multi-iterator object that has been previously “broadcasted,” finds the dimension with the smallest “sum of strides” in the broadcasted result and adapts all the iterators so as not to iterate over that dimension (by effectively making them of length-1 in that dimension). The corresponding dimension is returned unless mit->nd is 0, then -1 is returned. This function is useful for constructing ufunc-like routines that broadcast their inputs correctly and then call a strided 1-d version of the routine as the inner-loop. This 1-d version is usually optimized for speed and for this reason the loop should be performed over the axis that won’t require large stride jumps.

5.4.9 Neighborhood iterator

New in version 1.4.0. Neighborhood iterators are subclasses of the iterator object, and can be used to iterate over a neighborhood of a point. For example, you may want to iterate over every voxel of a 3d image, and for every such voxel, iterate over an hypercube. Neighborhood iterator automatically handle boundaries, thus making this kind of code much easier to write than manual boundaries handling, at the cost of a slight overhead.

PyObject* PyArray_NeighborhoodIterNew (PyArrayIterObject* iter, npy_intp bounds, int mode, PyArrayObject* fill_value)

This function creates a new neighborhood iterator from an existing iterator. The neighborhood will be computed.
relatively to the position currently pointed by `iter`, the bounds define the shape of the neighborhood iterator, and the mode argument the boundaries handling mode.

The `bounds` argument is expected to be a `(2 * iter->ao->nd)` arrays, such as the range `bound[2*i]->bounds[2*i+1]` defines the range where to walk for dimension `i` (both bounds are included in the walked coordinates). The bounds should be ordered for each dimension (`bounds[2*i] <= bounds[2*i+1]`).

The mode should be one of:

- `NPY_NEIGHBORHOOD_ITER_ZERO_PADDING`: zero padding. Outside bounds values will be 0.
- `NPY_NEIGHBORHOOD_ITER_ONE_PADDING`: one padding. Outside bounds values will be 1.
- `NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING`: constant padding. Outside bounds values will be the same as the first item in `fill_value`.
- `NPY_NEIGHBORHOOD_ITER_MIRROR_PADDING`: mirror padding. Outside bounds values will be as if the array items were mirrored. For example, for the array `[1, 2, 3, 4]`, `x[-2]` will be 2, `x[-2]` will be 1, `x[4]` will be 4, `x[5]` will be 1, etc...
- `NPY_NEIGHBORHOOD_ITER_CIRCULAR_PADDING`: circular padding. Outside bounds values will be as if the array was repeated. For example, for the array `[1, 2, 3, 4]`, `x[-2]` will be 3, `x[-2]` will be 4, `x[4]` will be 1, `x[5]` will be 2, etc...

If the mode is constant filling (`NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING`), `fill_value` should point to an array object which holds the filling value (the first item will be the filling value if the array contains more than one item). For other cases, `fill_value` may be NULL.

- The iterator holds a reference to `iter`
- Return NULL on failure (in which case the reference count of `iter` is not changed)
- `iter` itself can be a Neighborhood iterator: this can be useful for e.g automatic boundaries handling
- The object returned by this function should be safe to use as a normal iterator
- If the position of `iter` is changed, any subsequent call to `PyArrayNeighborhoodIter_Next` is undefined behavior, and `PyArrayNeighborhoodIter_Reset` must be called.

```python
PyArrayIterObject \*iter;
PyArrayNeighborhoodIterObject \*neigh_iter;
iter = PyArray_IterNew(x);

// For a 3x3 kernel
bounds = {-1, 1, -1, 1};
neigh_iter = (PyArrayNeighborhoodIterObject*)PyArrayNeighborhoodIter_New(
    iter, bounds, NPY_NEIGHBORHOOD_ITER_ZERO_PADDING, NULL);

for(i = 0; i < iter->size; ++i) {
    for (j = 0; j < neigh_iter->size; ++j) {
        // Walk around the item currently pointed by iter->dataptr
        PyArrayNeighborhoodIter_Next(neigh_iter);
    }
}

// Move to the next point of iter
PyArrayIter_Next(iter);
PyArrayNeighborhoodIter_Reset(neigh_iter);

int PyArrayNeighborhoodIter_Reset(PyArrayNeighborhoodIterObject* iter)
Reset the iterator position to the first point of the neighborhood. This should be called whenever the `iter` argument given at PyArray_NeighborhoodIterObject is changed (see example)
```
int PyArrayNeighborhoodIter_Next (PyArrayNeighborhoodIterObject* iter)
    After this call, iter->dataptr points to the next point of the neighborhood. Calling this function after every point
    of the neighborhood has been visited is undefined.

5.4.10 Array Scalars

PyObject* PyArray_Return (PyArrayObject* arr)
    This function checks to see if arr is a 0-dimensional array and, if so, returns the appropriate array scalar. It
    should be used whenever 0-dimensional arrays could be returned to Python.

PyObject* PyArray_Scalar (void* data, PyArray_Descr* dtype, PyObject* itemsize)
    Return an array scalar object of the given enumerated typenum and itemsize by copying from memory pointed
    to by data . If swap is nonzero then this function will byteswap the data if appropriate to the data-type because
    array scalars are always in correct machine-byte order.

PyObject* PyArray_ToScalar (void* data, PyArrayObject* arr)
    Return an array scalar object of the type and itemsize indicated by the array object arr copied from the memory
    pointed to by data and swapping if the data in arr is not in machine byte-order.

PyObject* PyArray_FromScalar (PyObject* scalar, PyArray_Descr* outcode)
    Return a 0-dimensional array of type determined by outcode from scalar which should be an array-scalar
    object. If outcode is NULL, then the type is determined from scalar.

void PyArray_ScalarAsCtype (PyObject* scalar, void* ctypeptr)
    Return in ctypeptr a pointer to the actual value in an array scalar. There is no error checking so scalar must be
    an array-scalar object, and ctypeptr must have enough space to hold the correct type. For flexible-sized types, a
    pointer to the data is copied into the memory of ctypeptr, for all other types, the actual data is copied into the
    address pointed to by ctypeptr.

void PyArray_CastScalarToCtype (PyObject* scalar, void* ctypeptr, PyArray_Descr* outcode)
    Return the data (cast to the data type indicated by outcode) from the array-scalar, scalar, into the memory
    pointed to by ctypeptr (which must be large enough to handle the incoming memory).

PyObject* PyArray_TypeObjectFromType (int type)
    Returns a scalar type-object from a type-number, type . Equivalent to PyArray_DescrFromType (type)-
    >typeobj except for reference counting and error-checking. Returns a new reference to the typeobject on
    success or NULL on failure.

NPY_SCALARKIND PyArray_ScalarKind (int typenum, PyArrayObject** arr)
    Return the kind of scalar represented by typenum and the array in *arr (if arr is not NULL ). The array is
    assumed to be rank-0 and only used if typenum represents a signed integer. If arr is not NULL and the first
    element is negative then NPY_INTNEG_SCALAR is returned, otherwise NPY_INTPOS_SCALAR is returned.
    The possible return values are NPY_{kind}_SCALAR where {kind} can be INTPOS, INTNEG, FLOAT,
    COMPLEX, BOOL, or OBJECT. NPY_NOSCALAR is also an enumerated value NPY_SCALARKIND
    variables can take on.

int PyArray_CanCoerceScalar (char thistype, char neededtype, NPY_SCALARKIND scalar)
    Implements the rules for scalar coercion. Scalars are only silently coerced from thistype to needed-
    type if this function returns nonzero. If scalar is NPY_NOSCALAR, then this function is equivalent to
    PyArray_CanCastSafely. The rule is that scalars of the same KIND can be coerced into arrays of the
    same KIND. This rule means that high-precision scalars will never cause low-precision arrays of the same
    KIND to be upcast.
5.4.11 Data-type descriptors

**Warning:** Data-type objects must be reference counted so be aware of the action on the data-type reference of different C-API calls. The standard rule is that when a data-type object is returned it is a new reference. Functions that take `PyArray_Descr *` objects and return arrays steal references to the data-type their inputs unless otherwise noted. Therefore, you must own a reference to any data-type object used as input to such a function.

```c
int PyArrayDescr_Check (PyObject* obj)
Evaluates as true if obj is a data-type object (PyArray_Descr *).

PyArray_Descr* PyArray_DescrNew (PyArray_Descr* obj)
Return a new data-type object copied from obj (the fields reference is just updated so that the new object points to the same fields dictionary if any).

PyArray_Descr* PyArray_DescrNewFromType (int typenum)
Create a new data-type object from the built-in (or user-registered) data-type indicated by typenum. All built-in types should not have any of their fields changed. This creates a new copy of the PyArray_Descr structure so that you can fill it in as appropriate. This function is especially needed for flexible data-types which need to have a new elsize member in order to be meaningful in array construction.

PyArray_Descr* PyArray_DescrNewByteorder (PyArray_Descr* obj, char newendian)
Create a new data-type object with the byteorder set according to newendian. All referenced data-type objects (in subdescr and fields members of the data-type object) are also changed (recursively). If a byteorder of NPY_IGNORE is encountered it is left alone. If newendian is NPY_SWAP, then all byte-orders are swapped. Other valid newendian values are NPY_NATIVE, NPY_LITTLE, and NPY_BIG which all cause the returned data-type descriptor (and all its referenced data-type descriptors) to have the corresponding byte-order.

PyArray_Descr* PyArray_DescrFromObject (PyObject* op, PyArray_Descr* mintype)
Determine an appropriate data-type object from the object op (which should be a “nested” sequence object) and the minimum data-type descriptor mintype (which can be NULL). Similar in behavior to array(op).dtype. Don’t confuse this function with PyArray_DescrConverter. This function essentially looks at all the objects in the (nested) sequence and determines the data-type from the elements it finds.

PyArray_Descr* PyArray_DescrFromScalar (PyObject* scalar)
Return a data-type object from an array-scalar object. No checking is done to be sure that scalar is an array scalar. If no suitable data-type can be determined, then a data-type of NPY_OBJECT is returned by default.

PyArray_Descr* PyArray_DescrFromType (int typenum)
Returns a data-type object corresponding to typenum. The typenum can be one of the enumerated types, a character code for one of the enumerated types, or a user-defined type.

int PyArray_DescrConverter (PyObject* obj, PyArray_Descr** dtype)
Convert any compatible Python object, obj, to a data-type object in dtype. A large number of Python objects can be converted to data-type objects. See Data type objects (dtype) for a complete description. This version of the converter converts None objects to a NPY_DEFAULT_TYPE data-type object. This function can be used with the “O&” character code in PyArg_ParseTuple processing.

int PyArray_DescrConverter2 (PyObject* obj, PyArray_Descr** dtype)
Convert any compatible Python object, obj, to a data-type object in dtype. This version of the converter converts None objects so that the returned data-type is NULL. This function can also be used with the “O&” character in PyArg_ParseTuple processing.

int Pyarray_DescrAlignConverter (PyObject* obj, PyArray_Descr** dtype)
Like PyArray_DescrConverter except it aligns C-struct-like objects on word-boundaries as the compiler would.
```
int Pyarray_DescrAlignConverter2 (PyObject* obj, PyArray_Descr** dtype)

Like PyArray_DescrConverter2 except it aligns C-struct-like objects on word-boundaries as the compiler would.

PyObject * PyArray_FieldNames (PyObject* dict)

Take the fields dictionary, dict, such as the one attached to a data-type object and construct an ordered-list of field names such as is stored in the names field of the PyArray_Descr object.

5.4.12 Conversion Utilities

For use with PyArg_ParseTuple

All of these functions can be used in PyArg_ParseTuple (...) with the “O&” format specifier to automatically convert any Python object to the required C-object. All of these functions return NPY_SUCCEED if successful and NPY_FAIL if not. The first argument to all of these function is a Python object. The second argument is the address of the C-type to convert the Python object to.

Warning: Be sure to understand what steps you should take to manage the memory when using these conversion functions. These functions can require freeing memory, and/or altering the reference counts of specific objects based on your use.

int PyArray_Converter (PyObject* obj, PyObject** address)

Convert any Python object to a PyArrayObject. If PyArray_Check (obj) is TRUE then its reference count is incremented and a reference placed in address. If obj is not an array, then convert it to an array using PyArray_FromAny. No matter what is returned, you must DECREF the object returned by this routine in address when you are done with it.

int PyArray_OutputConverter (PyObject* obj, PyArrayObject** address)

This is a default converter for output arrays given to functions. If obj is Py_None or NULL, then *address will be NULL but the call will succeed. If PyArray_Check (obj) is TRUE then it is returned in *address without incrementing its reference count.

int PyArray_IntpConverter (PyObject* obj, PyArray_Dims* seq)

Convert any Python sequence, obj, smaller than NPY_MAXDIMS to a C-array of npy_intp. The Python object could also be a single number. The seq variable is a pointer to a structure with members ptr and len. On successful return, seq ->ptr contains a pointer to memory that must be freed to avoid a memory leak. The restriction on memory size allows this converter to be conveniently used for sequences intended to be interpreted as array shapes.

int PyArray_BufferConverter (PyObject* obj, PyArray_Chunk* buf)

Convert any Python object, obj, with a (single-segment) buffer interface to a variable with members that detail the object’s use of its chunk of memory. The buf variable is a pointer to a structure with base, ptr, len, and flags members. The PyArray_Chunk structure is binary compatible with the Python’s buffer object (through its len member on 32-bit platforms and its ptr member on 64-bit platforms or in Python 2.5). On return, the base member is set to obj (or its base if obj is already a buffer object pointing to another object). If you need to hold on to the memory be sure to INCREF the base member. The chunk of memory is pointed to by buf ->ptr member and has length buf ->len. The flags member of buf is NPY_BEHAVED_RO with the NPY_WRITEABLE flag set if obj has a writeable buffer interface.

int PyArray_AxisConverter (PyObject * obj, int* axis)

Convert a Python object, obj, representing an axis argument to the proper value for passing to the functions that take an integer axis. Specifically, if obj is None, axis is set to NPY_MAXDIMS which is interpreted correctly by the C-API functions that take axis arguments.
int **PyArray_BoolConverter** *(PyObject* obj, Bool* value)*  
Convert any Python object, *obj*, to **NPY_TRUE** or **NPY_FALSE**, and place the result in *value*.

int **PyArray_ByteorderConverter** *(PyObject* obj, char* endian)*  

int **PyArray_SortkindConverter** *(PyObject* obj, NPY_SORTKIND* sort)*  
Convert Python strings into one of **NPY_QUICKSORT** (starts with ‘q’ or ‘Q’), **NPY_HEAPSORT** (starts with ‘h’ or ‘H’), or **NPY_MERGESORT** (starts with ‘m’ or ‘M’).

int **PyArray_SearchsideConverter** *(PyObject* obj, NPY_SEARCHSIDE* side)*  
Convert Python strings into one of **NPY_SEARCHLEFT** (starts with ‘l’ or ‘L’), or **NPY_SEARCHRIGHT** (starts with ‘r’ or ‘R’).

**Other conversions**

int **PyArray_PyIntAsInt** *(PyObject* op)*  
Convert all kinds of Python objects (including arrays and array scalars) to a standard integer. On error, -1 is returned and an exception set. You may find useful the macro:

    #define error_converting(x) (((x) == -1) && PyErr_Occurred())

**PyArray_PyIntAsIntp** *(PyObject* op)*  
Convert all kinds of Python objects (including arrays and array scalars) to a (platform-pointer-sized) integer. On error, -1 is returned and an exception set.

int **PyArray_IntpFromSequence** *(PyObject* seq, npy_intp* vals, int maxvals)*  
Convert any Python sequence (or single Python number) passed in as *seq* to (up to) *maxvals* pointer-sized integers and place them in the *vals* array. The sequence can be smaller then *maxvals* as the number of converted objects is returned.

int **PyArray_TypestrConvert** *(int itemsize, int gentype)*  
Convert typestring characters (with *itemsize*) to basic enumerated data types. The typestring character corresponding to signed and unsigned integers, floating point numbers, and complex-floating point numbers are recognized and converted. Other values of gentype are returned. This function can be used to convert, for example, the string ‘f4’ to **NPY_FLOAT32**.

### 5.4.13 Miscellaneous

**Importing the API**

In order to make use of the C-API from another extension module, the **import_array** () command must be used. If the extension module is self-contained in a single .c file, then that is all that needs to be done. If, however, the extension module involves multiple files where the C-API is needed then some additional steps must be taken.

```c
void import_array(void)
```

This function must be called in the initialization section of a module that will make use of the C-API. It imports the module where the function-pointer table is stored and points the correct variable to it.

**PY_ARRAY_UNIQUE_SYMBOL**

**NO_IMPORT_ARRAY**

Using these #defines you can use the C-API in multiple files for a single extension module. In each file you must define **PY_ARRAY_UNIQUE_SYMBOL** to some name that will hold the C-API (e.g. myextension_ARRAY_API). This must be done before including the numpy/arrayobject.h file. In the module...
initialization routine you call import_array(). In addition, in the files that do not have the module initialization sub_routine define NO_IMPORT_ARRAY prior to including numpy/arrayobject.h.

Suppose I have two files coolmodule.c and coolhelper.c which need to be compiled and linked into a single extension module. Suppose coolmodule.c contains the required initcool module initialization function (with the import_array() function called). Then, coolmodule.c would have at the top:

```c
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

On the other hand, coolhelper.c would contain at the top:

```c
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#define NO_IMPORT_ARRAY
#include numpy/arrayobject.h
```

**Checking the API Version**

Because python extensions are not used in the same way as usual libraries on most platforms, some errors cannot be automatically detected at build time or even runtime. For example, if you build an extension using a function available only for numpy ≥ 1.3.0, and you import the extension later with numpy 1.2, you will not get an import error (but almost certainly a segmentation fault when calling the function). That’s why several functions are provided to check for numpy versions. The macros NPY_VERSION and NPY_FEATURE_VERSION corresponds to the numpy version used to build the extension, whereas the versions returned by the functions PyArray_GetNDArrayCVersion and PyArray_GetNDArrayCFeatureVersion corresponds to the runtime numpy’s version.

The rules for ABI and API compatibilities can be summarized as follows:

- Whenever NPY_VERSION != PyArray_GetNDArrayCVersion, the extension has to be recompiled (ABI incompatibility).
- NPY_VERSION == PyArray_GetNDArrayCVersion and NPY_FEATURE_VERSION <= PyArray_GetNDArrayCFeatureVersion means backward compatible changes.

ABI incompatibility is automatically detected in every numpy’s version. API incompatibility detection was added in numpy 1.4.0. If you want to supported many different numpy versions with one extension binary, you have to build your extension with the lowest NPY_FEATURE_VERSION as possible.

```c
unsigned int PyArray_GetNDArrayCVersion(void)
```

This just returns the value NPY_VERSION. NPY_VERSION changes whenever a backward incompatible change at the ABI level. Because it is in the C-API, however, comparing the output of this function from the value defined in the current header gives a way to test if the C-API has changed thus requiring a re-Compilation of extension modules that use the C-API. This is automatically checked in the function import_array.

```c
unsigned int PyArray_GetNDArrayCFeatureVersion(void)
```

New in version 1.4.0. This just returns the value NPY_FEATURE_VERSION. NPY_FEATURE_VERSION changes whenever the API changes (e.g. a function is added). A changed value does not always require a recompile.

**Internal Flexibility**

```c
int PyArray_SetNumericOps(PyObject* dict)
```

NumPy stores an internal table of Python callable objects that are used to implement arithmetic operations for arrays as well as certain array calculation methods. This function allows the user to replace any or all of these Python objects with their own versions. The keys of the dictionary, dict, are the named functions to replace and the paired value is the Python callable object to use. Care should be taken that the function used to replace an
internal array operation does not itself call back to that internal array operation (unless you have designed the function to handle that), or an unchecked infinite recursion can result (possibly causing program crash). The key names that represent operations that can be replaced are:

- `add`
- `subtract`
- `multiply`
- `divide`
- `remainder`
- `power`
- `square`
- `reciprocal`
- `ones_like`
- `sqrt`
- `negative`
- `absolute`
- `invert`
- `left_shift`
- `right_shift`
- `bitwise_and`
- `bitwise_xor`
- `bitwise_or`
- `less`
- `less_equal`
- `equal`
- `not_equal`
- `greater`
- `greater_equal`
- `floor_divide`
- `true_divide`
- `logical_or`
- `logical_and`
- `floor`
- `ceil`
- `maximum`
- `minimum`
- `rint`.

These functions are included here because they are used at least once in the array object’s methods. The function returns -1 (without setting a Python Error) if one of the objects being assigned is not callable.

```python
PyObject* PyArray_GetNumericOps (void)
Return a Python dictionary containing the callable Python objects stored in the the internal arithmetic operation table. The keys of this dictionary are given in the explanation for PyArray_SetNumericOps.
```

```python
void PyArray_SetStringFunction (PyObject* op, int repr)
This function allows you to alter the tp_str and tp_repr methods of the array object to any Python function. Thus you can alter what happens for all arrays when str(arr) or repr(arr) is called from Python. The function to be called is passed in as op. If repr is non-zero, then this function will be called in response to repr(arr), otherwise the function will be called in response to str(arr). No check on whether or not op is callable is performed. The callable passed in to op should expect an array argument and should return a string to be printed.
```

## Memory management

```python
char* PyDataMem_NEW (size_t nbytes)
```

```python
PyDataMem_FREE (char* ptr)
```

```python
char* PyDataMem_RENEW (void* ptr, size_t newbytes)
Macros to allocate, free, and reallocate memory. These macros are used internally to create arrays.
```

```python
npy_intp* PyDimMem_NEW (nd)
```

```python
PyDimMem_FREE (npy_intp* ptr)
```

```python
npy_intp* PyDimMem_RENEW (npy_intp* ptr, npy_intp newnd)
Macros to allocate, free, and reallocate dimension and strides memory.
```

```python
PyArray_malloc (nbytes)
```

```python
PyArray_free (ptr)
```

```python
PyArray_realloc (ptr, nbytes)
These macros use different memory allocators, depending on the constant NPY_USE_PYMEM. The system malloc is used when NPY_USE_PYMEM is 0, if NPY_USE_PYMEM is 1, then the Python memory allocator is used.
```

## Threading support

These macros are only meaningful if NPY_ALLOW_THREADS evaluates True during compilation of the extension module. Otherwise, these macros are equivalent to whitespace. Python uses a single Global Interpreter Lock (GIL) for each Python process so that only a single thread may execute at a time (even on multi-cpu machines). When
calling out to a compiled function that may take time to compute (and does not have side-effects for other threads like updated global variables), the GIL should be released so that other Python threads can run while the time-consuming calculations are performed. This can be accomplished using two groups of macros. Typically, if one macro in a group is used in a code block, all of them must be used in the same code block. Currently, \texttt{NPY\_ALLOW\_THREADS} is defined to the python-defined \texttt{WITH\_THREADS} constant unless the environment variable \texttt{NPY\_NOSMP} is set in which case \texttt{NPY\_ALLOW\_THREADS} is defined to be 0.

\textbf{Group 1}

This group is used to call code that may take some time but does not use any Python C-API calls. Thus, the GIL should be released during its calculation.

\texttt{NPY\_BEGIN\_ALLOW\_THREADS}

Equivalent to \texttt{Py\_BEGIN\_ALLOW\_THREADS} except it uses \texttt{NPY\_ALLOW\_THREADS} to determine if the macro if replaced with white-space or not.

\texttt{NPY\_END\_ALLOW\_THREADS}

Equivalent to \texttt{Py\_END\_ALLOW\_THREADS} except it uses \texttt{NPY\_ALLOW\_THREADS} to determine if the macro if replaced with white-space or not.

\texttt{NPY\_BEGIN\_THREADS\_DEF}

Place in the variable declaration area. This macro sets up the variable needed for storing the Python state.

\texttt{NPY\_BEGIN\_THREADS}

Place right before code that does not need the Python interpreter (no Python C-API calls). This macro saves the Python state and releases the GIL.

\texttt{NPY\_END\_THREADS}

Place right after code that does not need the Python interpreter. This macro acquires the GIL and restores the Python state from the saved variable.

\texttt{NPY\_BEGIN\_THREADS\_DESCR} (\texttt{PyArray\_Descr *dtype})

Useful to release the GIL only if \texttt{dtype} does not contain arbitrary Python objects which may need the Python interpreter during execution of the loop. Equivalent to

\texttt{NPY\_END\_THREADS\_DESCR} (\texttt{PyArray\_Descr *dtype})

Useful to regain the GIL in situations where it was released using the BEGIN form of this macro.

\textbf{Group 2}

This group is used to re-acquire the Python GIL after it has been released. For example, suppose the GIL has been released (using the previous calls), and then some path in the code (perhaps in a different subroutine) requires use of the Python C-API, then these macros are useful to acquire the GIL. These macros accomplish essentially a reverse of the previous three (acquire the LOCK saving what state it had) and then re-release it with the saved state.

\texttt{NPY\_ALLOW\_C\_API\_DEF}

Place in the variable declaration area to set up the necessary variable.

\texttt{NPY\_ALLOW\_C\_API}

Place before code that needs to call the Python C-API (when it is known that the GIL has already been released).

\texttt{NPY\_DISABLE\_C\_API}

Place after code that needs to call the Python C-API (to re-release the GIL).

\textbf{Tip:} Never use semicolons after the threading support macros.
Priority

NPY_PRIORITY
Default priority for arrays.

NPY_SUBTYPE_PRIORITY
Default subtype priority.

NPY_SCALAR_PRIORITY
Default scalar priority (very small)

double PyArray_GetPriority(PyObject* obj, double def)
Return the __array_priority__ attribute (converted to a double) of obj or def if no attribute of that name exists. Fast returns that avoid the attribute lookup are provided for objects of type PyArray_Type.

Default buffers

NPY_BUFSIZE
Default size of the user-settable internal buffers.

NPY_MIN_BUFSIZE
Smallest size of user-settable internal buffers.

NPY_MAX_BUFSIZE
Largest size allowed for the user-settable buffers.

Other constants

NPY_NUM_FLOATTYPE
The number of floating-point types

NPY_MAXDIMS
The maximum number of dimensions allowed in arrays.

NPY_VERSION
The current version of the ndarray object (check to see if this variable is defined to guarantee the numpy/arrayobject.h header is being used).

NPY_FALSE
Defined as 0 for use with Bool.

NPY_TRUE
Defined as 1 for use with Bool.

NPY_FAIL
The return value of failed converter functions which are called using the “O&” syntax in PyArg_ParseTuple-like functions.

NPY_SUCCEED
The return value of successful converter functions which are called using the “O&” syntax in PyArg_ParseTuple-like functions.

Miscellaneous Macros

PyArray_SAMESHAPE(a1, a2)
Evaluates as True if arrays a1 and a2 have the same shape.
**PyArray_MAX** \((a, b)\)

Returns the maximum of \(a\) and \(b\). If \((a)\) or \((b)\) are expressions they are evaluated twice.

**PyArray_MIN** \((a, b)\)

Returns the minimum of \(a\) and \(b\). If \((a)\) or \((b)\) are expressions they are evaluated twice.

**PyArray_CLT** \((a, b)\)

**PyArray_CGT** \((a, b)\)

**PyArray_CLE** \((a, b)\)

**PyArray_CGE** \((a, b)\)

**PyArray_CEQ** \((a, b)\)

**PyArray_CNE** \((a, b)\)

Implements the complex comparisons between two complex numbers (structures with a real and imag member) using NumPy’s definition of the ordering which is lexicographic: comparing the real parts first and then the complex parts if the real parts are equal.

**PyArray_REFCOUNT** \((PyObject* \ op)\)

Returns the reference count of any Python object.

**PyArray_XDECREF_ERR** \((PyObject *\ obj)\)

DECREF’s an array object which may have the NPY_UPDATEIFCOPY flag set without causing the contents to be copied back into the original array. Resets the NPY_WRITEABLE flag on the base object. This is useful for recovering from an error condition when NPY_UPDATEIFCOPY is used.

### Enumerated Types

**NPY_SORTKIND**

A special variable-type which can take on the values NPY_{KIND} where {KIND} is

- QUICKSORT, HEAPSORT, MERGESORT

**NPY_NSORTS**

Defined to be the number of sorts.

**NPY_SCALARKIND**

A special variable type indicating the number of “kinds” of scalars distinguished in determining scalar-coercion rules. This variable can take on the values NPY_{KIND} where {KIND} can be

- NOSCALAR, BOOL_SCALAR, INTPOS_SCALAR, INTNEG_SCALAR, FLOAT_SCALAR, COMPLEX_SCALAR, OBJECT_SCALAR

**NPY_NSCALARKINDS**

Defined to be the number of scalar kinds (not including NPY_NOSCALAR).

**NPY_ORDER**

A variable type indicating the order that an array should be interpreted in. The value of a variable of this type can be NPY_{ORDER} where {ORDER} is

- ANYORDER, CORDER, FORTRANORDER
NPY_CLIPMODE
A variable type indicating the kind of clipping that should be applied in certain functions. The value of a
variable of this type can be NPY_{MODE} where {MODE} is

CLIP, WRAP, RAISE

5.5 UFunc API

5.5.1 Constants

UFUNC_ERR_{HANDLER}
{HANDLER} can be IGNORE, WARN, RAISE, or CALL

UFUNC_{THING}_{ERR}
{THING} can be MASK, SHIFT, or FPE, and {ERR} can be DIVIDE BY ZERO, OVERFLOW, UNDERFLOW, and INVALID.

PyUFunc_{VALUE}
{VALUE} can be One (1), Zero (0), or None (-1)

5.5.2 Macros

NPY_LOOP_BEGIN_THREADS
Used in universal function code to only release the Python GIL if loop->obj is not true (i.e. this is not an
OBJECT array loop). Requires use of NPY_BEGIN_THREADS_DEF in variable declaration area.

NPY_LOOP_END_THREADS
Used in universal function code to re-acquire the Python GIL if it was released (because loop->obj was not
true).

UFUNC_CHECK_ERROR (loop)
A macro used internally to check for errors and goto fail if found. This macro requires a fail label in the current
code block. The loop variable must have at least members (obj, errmask, and errorbj). If loop ->obj is nonzero, then PyErr_Occurred () is called (meaning the GIL must be held). If loop ->obj is zero, then if
loop ->errormask is nonzero, PyUFunc_checkfperr is called with arguments loop ->errormask and loop
->errobj. If the result of this check of the IEEE floating point registers is true then the code redirects to the fail
label which must be defined.

UFUNC_CHECK_STATUS (ret)
A macro that expands to platform-dependent code. The ret variable can be any integer. The
UFUNC_FPE_{ERR} bits are set in ret according to the status of the corresponding error flags of the
floating point processor.

5.5.3 Functions

PyObject* PyUFunc_FromFuncAndData (PyUFuncGenericFunction* func, void** data, char* types, int
ntypes, int nin, int nout, int identity, char* name, char* doc, int
check_return)

Create a new broadcasting universal function from required variables. Each ufunc builds around the notion of
an element-by-element operation. Each ufunc object contains pointers to 1-d loops implementing the basic
functionality for each supported type.

Note: The func, data, types, name, and doc arguments are not copied by PyUFunc_FromFuncAndData.
The caller must ensure that the memory used by these arrays is not freed as long as the ufunc object is alive.
Parameters

- `func` – Must be an array of length `ntypes` containing `PyUFuncGenericFunction` items. These items are pointers to functions that actually implement the underlying (element-by-element) function `N` times.

- `data` – Should be NULL or a pointer to an array of size `ntypes`. This array may contain arbitrary extra-data to be passed to the corresponding 1-d loop function in the `func` array.

- `types` – Must be of length `(nin + nout) * ntypes`, and it contains the data-types (built-in only) that the corresponding function in the `func` array can deal with.

- `ntypes` – How many different data-type “signatures” the ufunc has implemented.

- `nin` – The number of inputs to this operation.

- `nout` – The number of outputs

- `identity` – XXX: Undocumented

- `name` – The name for the ufunc. Specifying a name of ‘add’ or ‘multiply’ enables a special behavior for integer-typed reductions when no `dtype` is given. If the input type is an integer (or boolean) data type smaller than the size of the `int_` data type, it will be internally upcast to the `int_` (or `uint`) data type.

- `doc` – Allows passing in a documentation string to be stored with the ufunc. The documentation string should not contain the name of the function or the calling signature as that will be dynamically determined from the object and available when accessing the `__doc__` attribute of the ufunc.

- `check_return` – Unused and present for backwards compatibility of the C-API. A corresponding `check_return` integer does exist in the ufunc structure and it does get set with this value when the ufunc object is created.

```c
int PyUFunc_RegisterLoopForType(PyUFuncObject* ufunc, int usertype, PyUFuncGenericFunction function, int* arg_types, void* data)
```

This function allows the user to register a 1-d loop with an already-created ufunc to be used whenever the ufunc is called with any of its input arguments as the user-defined data-type. This is needed in order to make ufuncs work with built-in data-types. The data-type must have been previously registered with the numpy system. The loop is passed in as `function`. This loop can take arbitrary data which should be passed in as `data`. The data-types the loop requires are passed in as `arg_types` which must be a pointer to memory at least as large as `ufunc->nargs`.

```c
int PyUFunc_ReplaceLoopBySignature(PyUFuncObject* ufunc, PyUFuncGenericFunction newfunc, int* signature, PyUFuncGenericFunction* oldfunc)
```

Replace a 1-d loop matching the given `signature` in the already-created `ufunc` with the new 1-d loop `newfunc`. Return the old 1-d loop function in `oldfunc`. Return 0 on success and -1 on failure. This function works only with built-in types (use `PyUFunc_RegisterLoopForType` for user-defined types). A signature is an array of data-type numbers indicating the inputs followed by the outputs assumed by the 1-d loop.

```c
int PyUFunc_GenericFunction(PyUFuncObject* self, PyObject* args, PyArrayObject** mps)
```

A generic ufunc call. The ufunc is passed in as `self`, the arguments to the ufunc as `args`. The `mps` argument is an array of `PyArrayObject` pointers containing the converted input arguments as well as the ufunc outputs on return. The user is responsible for managing this array and receives a new reference for each array in `mps`. The total number of arrays in `mps` is given by `self->nin + self->nout`.

```c
int PyUFunc_checkfperr(int errmask, PyObject* errobj)
```

A simple interface to the IEEE error-flag checking support. The `errmask` argument is a mask of `UFUNC_MASK_{ERR}` bitmasks indicating which errors to check for (and how to check for them). The `errobj` must be a Python tuple with two elements: a string containing the name which will be used in any...
communication of error and either a callable Python object (call-back function) or **Py_None**. The callable object will only be used if **UFUNC_ERR_CALL** is set as the desired error checking method. This routine manages the GIL and is safe to call even after releasing the GIL. If an error in the IEEE-compatible hardware is determined a -1 is returned, otherwise a 0 is returned.

```c
void PyUFunc_clearfperr()
Clear the IEEE error flags.

void PyUFunc_GetPyValues(char* name, int* bufsize, int* errmask, PyObject** errobj)
Get the Python values used for ufunc processing from the thread-local storage area unless the defaults have been set in which case the name lookup is bypassed. The name is placed as a string in the first element of *errobj. The second element is the looked-up function to call on error callback. The value of the looked-up buffer-size to use is passed into bufsize, and the value of the error mask is placed into errmask.
```

### 5.5.4 Generic functions

At the core of every ufunc is a collection of type-specific functions that defines the basic functionality for each of the supported types. These functions must evaluate the underlying function \( N \geq 1 \) times. Extra-data may be passed in that may be used during the calculation. This feature allows some general functions to be used as these basic looping functions. The general function has all the code needed to point variables to the right place and set up a function call. The general function assumes that the actual function to call is passed in as the extra data and calls it with the correct values. All of these functions are suitable for placing directly in the array of functions stored in the functions member of the PyUFuncObject structure.

```c
void PyUFunc_f_f_As_d_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_d_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_f_f(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_g_g(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_F_F_As_D_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_F_F(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_D_D(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_G_G(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking one input argument and returning one output. This function is passed in func. The letters correspond to dtypechar’s of the supported data types (\( f \) - float, \( d \) - double, \( g \) - long double, \( F \) - cfloat, \( D \) - cdouble, \( G \) - clongdouble). The argument func must support the same signature. The _As_X_X variants assume ndarray’s of one data type but cast the values to use an underlying function that takes a different data type. Thus, **PyUFunc_f_f_As_d_d** uses ndarrays of data type **NPY_FLOAT** but calls out to a C-function that takes double and returns double.

```c
void PyUFunc_ff_f_As_dd_d(char** args, npy_intp* dimensions, npy_intp* steps, void* func)

void PyUFunc_ff_f(char** args, npy_intp* dimensions, npy_intp* steps, void* func)
```
void \texttt{PyUFunc\_dd\_d}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

void \texttt{PyUFunc\_gg\_g}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

void \texttt{PyUFunc\_FF\_F\_As\_DD\_D}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

void \texttt{PyUFunc\_DD\_D}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

void \texttt{PyUFunc\_FF\_F}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

void \texttt{PyUFunc\_GG\_G}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking two input arguments and returning one output. The underlying function to call is passed in as \textit{func}. The letters correspond to dtypechar's of the specific data type supported by the general-purpose function. The argument \textit{func} must support the corresponding signature. The \_As\_XX\_X variants assume ndarrays of one data type but cast the values at each iteration of the loop to use the underlying function that takes a different data type.

void \texttt{PyUFunc\_O\_O}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

void \texttt{PyUFunc\_OO\_O}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

One-input, one-output, and two-input, one-output core 1-d functions for the NPY\_OBJECT data type. These functions handle reference count issues and return early on error. The actual function to call is \textit{func} and it must accept calls with the signature (PyObject*) (PyObject*) for \texttt{PyUFunc\_O\_O} or (PyObject*) (PyObject *, PyObject *) for \texttt{PyUFunc\_OO\_O}.

void \texttt{PyUFunc\_O\_O\_method}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

This general purpose 1-d core function assumes that \textit{func} is a string representing a method of the input object. For each iteration of the loop, the Python object is extracted from the array and its \textit{func} method is called returning the result to the output array.

void \texttt{PyUFunc\_OO\_O\_method}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

This general purpose 1-d core function assumes that \textit{func} is a string representing a method of the input object that takes one argument. The first argument in \textit{args} is the method whose function is called, the second argument in \textit{args} is the argument passed to the function. The output of the function is stored in the third entry of \textit{args}.

void \texttt{PyUFunc\_On\_Om}(char** \textit{args}, npy\_intp* \textit{dimensions}, npy\_intp* \textit{steps}, void* \textit{func})

This is the 1-d core function used by the dynamic ufuncs created by umath.frompyfunc(function, nin, nout). In this case \textit{func} is a pointer to a PyUFunc\_PyFuncData structure which has definition

\begin{verbatim}
typedef struct {
    int nin;
    int nout;
    PyObject *callable;
} PyUFunc_PyFuncData;
\end{verbatim}

At each iteration of the loop, the \textit{nin} input objects are extracted from their object arrays and placed into an argument tuple, the Python \textit{callable} is called with the input arguments, and the \textit{nout} outputs are placed into their object arrays.

5.5. UFunc API
5.5.5 Importing the API

PY_UFUNC_UNIQUE_SYMBOL

NO_IMPORT_UFUNC

void import_ufunc(void)

These are the constants and functions for accessing the ufunc C-API from extension modules in precisely the same way as the array C-API can be accessed. The import_ufunc() function must always be called (in the initialization subroutine of the extension module). If your extension module is in one file then that is all that is required. The other two constants are useful if your extension module makes use of multiple files. In that case, define PY_UFUNC_UNIQUE_SYMBOL to something unique to your code and then in source files that do not contain the module initialization function but still need access to the UFUNC API, define PY_UFUNC_UNIQUE_SYMBOL to the same name used previously and also define NO_IMPORT_UFUNC.

The C-API is actually an array of function pointers. This array is created (and pointed to by a global variable) by import_ufunc. The global variable is either statically defined or allowed to be seen by other files depending on the state of PY_UFUNC_UNIQUE_SYMBOL and NO_IMPORT_UFUNC.

5.6 Generalized Universal Function API

There is a general need for looping over not only functions on scalars but also over functions on vectors (or arrays), as explained on http://scipy.org/scipy/numpy/wiki/GeneralLoopingFunctions. We propose to realize this concept by generalizing the universal functions (ufuncs), and provide a C implementation that adds ~500 lines to the numpy code base. In current (specialized) ufuncs, the elementary function is limited to element-by-element operations, whereas the generalized version supports “sub-array” by “sub-array” operations. The Perl vector library PDL provides a similar functionality and its terms are re-used in the following.

Each generalized ufunc has information associated with it that states what the “core” dimensionality of the inputs is, as well as the corresponding dimensionality of the outputs (the element-wise ufuncs have zero core dimensions). The list of the core dimensions for all arguments is called the “signature” of a ufunc. For example, the ufunc numpy.add has signature (), ()->() defining two scalar inputs and one scalar output.

Another example is (see the GeneralLoopingFunctions page) the function inner1d(a,b) with a signature of (i), (i)->(). This applies the inner product along the last axis of each input, but keeps the remaining indices intact. For example, where a is of shape (3,5,N) and b is of shape (5,N), this will return an output of shape (3,5). The underlying elementary function is called 3*5 times. In the signature, we specify one core dimension (i) for each input and zero core dimensions () for the output, since it takes two 1-d arrays and returns a scalar. By using the same name i, we specify that the two corresponding dimensions should be of the same size (or one of them is of size 1 and will be broadcasted).

The dimensions beyond the core dimensions are called “loop” dimensions. In the above example, this corresponds to (3,5).

The usual numpy “broadcasting” rules apply, where the signature determines how the dimensions of each input/output object are split into core and loop dimensions:

1. While an input array has a smaller dimensionality than the corresponding number of core dimensions, 1’s are pre-pended to its shape.
2. The core dimensions are removed from all inputs and the remaining dimensions are broadcasted; defining the loop dimensions.
3. The output is given by the loop dimensions plus the output core dimensions.
5.6.1 Definitions

Elementary Function
Each ufunc consists of an elementary function that performs the most basic operation on the smallest portion of array arguments (e.g. adding two numbers is the most basic operation in adding two arrays). The ufunc applies the elementary function multiple times on different parts of the arrays. The input/output of elementary functions can be vectors; e.g., the elementary function of inner1d takes two vectors as input.

Signature
A signature is a string describing the input/output dimensions of the elementary function of a ufunc. See section below for more details.

Core Dimension
The dimensionality of each input/output of an elementary function is defined by its core dimensions (zero core dimensions correspond to a scalar input/output). The core dimensions are mapped to the last dimensions of the input/output arrays.

Dimension Name
A dimension name represents a core dimension in the signature. Different dimensions may share a name, indicating that they are of the same size (or are broadcastable).

Dimension Index
A dimension index is an integer representing a dimension name. It enumerates the dimension names according to the order of the first occurrence of each name in the signature.

5.6.2 Details of Signature

The signature defines “core” dimensionality of input and output variables, and thereby also defines the contraction of the dimensions. The signature is represented by a string of the following format:

- Core dimensions of each input or output array are represented by a list of dimension names in parentheses, \((i_1, \ldots, i_N)\); a scalar input/output is denoted by \(()\). Instead of \(i_1, i_2\), etc, one can use any valid Python variable name.
- Dimension lists for different arguments are separated by ",". Input/output arguments are separated by "->".
- If one uses the same dimension name in multiple locations, this enforces the same size (or broadcastable size) of the corresponding dimensions.

The formal syntax of signatures is as follows:

\[
\begin{align*}
\text{Signature} & ::= \text{Input arguments} "->" \text{Output arguments} \\
\text{Input arguments} & ::= \text{Argument list} \\
\text{Output arguments} & ::= \text{Argument list} \\
\text{Argument list} & ::= \text{nil} | \text{Argument} | \text{Argument} "," \text{Argument list} \\
\text{Argument} & ::= "(" \text{Core dimension list} ")" \\
\text{Core dimension list} & ::= \text{nil} | \text{Dimension name} | \text{Dimension name} "," \text{Core dimension list} \\
\text{Dimension name} & ::= \text{valid Python variable name}
\end{align*}
\]

Notes:
1. All quotes are for clarity.
2. Core dimensions that share the same name must be broadcastable, as the two \(i\) in our example above. Each dimension name typically corresponding to one level of looping in the elementary function’s implementation.
3. White spaces are ignored.
Here are some examples of signatures:

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>() -&gt; ()</td>
<td></td>
</tr>
<tr>
<td>inner1d</td>
<td>(i) -&gt; ()</td>
<td></td>
</tr>
<tr>
<td>sum1d</td>
<td>(i) -&gt; ()</td>
<td></td>
</tr>
<tr>
<td>dot2d</td>
<td>(m,n), (n,p) -&gt; (m,p)</td>
<td>matrix multiplication</td>
</tr>
<tr>
<td>outer_inner</td>
<td>(i,t), (j,t) -&gt; (i,j)</td>
<td>inner over the last dimension, outer over the second to last, and loop/broadcast over the rest.</td>
</tr>
</tbody>
</table>

### 5.6.3 C-API for implementing Elementary Functions

The current interface remains unchanged, and `PyUFunc_FromFuncAndData` can still be used to implement (specialized) ufuncs, consisting of scalar elementary functions. One can use `PyUFunc_FromFuncAndDataAndSignature` to declare a more general ufunc. The argument list is the same as `PyUFunc_FromFuncAndData`, with an additional argument specifying the signature as C string.

Furthermore, the callback function is of the same type as before, void (*foo)(char **args, intp *dimensions, intp *steps, void *func). When invoked, `args` is a list of length `nargs` containing the data of all input/output arguments. For a scalar elementary function, `steps` is also of length `nargs`, denoting the strides used for the arguments. `dimensions` is a pointer to a single integer defining the size of the axis to be looped over.

For a non-trivial signature, `dimensions` will also contain the sizes of the core dimensions as well, starting at the second entry. Only one size is provided for each unique dimension name and the sizes are given according to the first occurrence of a dimension name in the signature.

The first `nargs` elements of `steps` remain the same as for scalar ufuncs. The following elements contain the strides of all core dimensions for all arguments in order.

For example, consider a ufunc with signature (i,j), (i) -> (). In this case, `args` will contain three pointers to the data of the input/output arrays `a`, `b`, `c`. Furthermore, `dimensions` will be [N, I, J] to define the size of N of the loop and the sizes I and J for the core dimensions i and j. Finally, `steps` will be [a_N, b_N, c_N, a_i, a_j, b_i], containing all necessary strides.

### 5.7 Numpy core libraries

New in version 1.3.0. Starting from numpy 1.3.0, we are working on separating the pure C, “computational” code from the python dependent code. The goal is twofolds: making the code cleaner, and enabling code reuse by other extensions outside numpy (scipy, etc...).

#### 5.7.1 Numpy core math library

The numpy core math library (‘npymath’) is a first step in this direction. This library contains most math-related C99 functionality, which can be used on platforms where C99 is not well supported. The core math functions have the same API as the C99 ones, except for the `npy_` prefix.

The available functions are defined in `npy_math.h` - please refer to this header in doubt.

### Floating point classification

**NPY_NAN**

This macro is defined to a NaN (Not a Number), and is guaranteed to have the signbit unset (‘positive’ NaN).
The corresponding single and extension precision macro are available with the suffix F and L.

**NPY_INFINITY**
The corresponding single and extension precision macro are available with the suffix F and L.

**NPY_PZERO**
The corresponding single and extension precision macro are available with the suffix F and L.

**NPY_NZERO**
The corresponding single and extension precision macro are available with the suffix F and L.

```c
int npy_isnan(x)
This is a macro, and is equivalent to C99 isnan: works for single, double and extended precision, and return a
non 0 value is x is a NaN.
```

```c
int npy_isfinite(x)
This is a macro, and is equivalent to C99 isfinite: works for single, double and extended precision, and return a
non 0 value is x is neither a NaN or a infinity.
```

```c
int npy_isinf(x)
This is a macro, and is equivalent to C99 isinf: works for single, double and extended precision, and return a
non 0 value is x is infinite (positive and negative).
```

```c
int npy_signbit(x)
This is a macro, and is equivalent to C99 signbit: works for single, double and extended precision, and return a
non 0 value is x has the signbit set (that is the number is negative).
```

```c
double npy_copysign(double x, double y)
This is a function equivalent to C99 copysign: return x with the same sign as y. Works for any value, including
inf and nan. Single and extended precisions are available with suffix f and l. New in version 1.4.0.
```

### Useful math constants

The following math constants are available in npy_math.h. Single and extended precision are also available by adding
the F and L suffixes respectively.

**NPY_E**
- Base of natural logarithm (e)

**NPY_LOG2E**
- Logarithm to base 2 of the Euler constant \( \frac{\ln(e)}{\ln(2)} \)

**NPY_LOG10E**
- Logarithm to base 10 of the Euler constant \( \frac{\ln(e)}{\ln(10)} \)

**NPY_LOGE2**
- Natural logarithm of 2 (ln(2))

**NPY_LOGE10**
- Natural logarithm of 10 (ln(10))

**NPY_PI**
- Pi (\( \pi \))

**NPY_PI_2**
- Pi divided by 2 (\( \frac{\pi}{2} \))
**NPY_PI_4**
Pi divided by 4 ($\frac{\pi}{4}$)

**NPY_1_PI**
Reciprocal of pi ($\frac{1}{\pi}$)

**NPY_2_PI**
Two times the reciprocal of pi ($\frac{2}{\pi}$)

**NPY_EULER**
The Euler constant ($\lim_{n \to \infty} \sum_{k=1}^{n} \frac{1}{k} - \ln n$)

**Low-level floating point manipulation**

Those can be useful for precise floating point comparison.

```c
double npy_nextafter(double x, double y)
```
This is a function equivalent to C99 nextafter: return next representable floating point value from x in the direction of y. Single and extended precisions are available with suffix f and l. New in version 1.4.0.

```c
double npy_spacing(double x)
```
This is a function equivalent to Fortran intrinsic. Return distance between x and next representable floating point value from x, e.g. spacing(1) == eps. spacing of nan and +/- inf return nan. Single and extended precisions are available with suffix f and l. New in version 1.4.0.

**Complex functions**

New in version 1.4.0. C99-like complex functions have been added. Those can be used if you wish to implement portable C extensions. Since we still support platforms without C99 complex type, you need to restrict to C90-compatible syntax, e.g.:

```c
/* a = 1 + 2i */
npy_complex a = npy_cpack(1, 2);
npy_complex b;

b = npy_log(a);
```

**Linking against the core math library in an extension**

New in version 1.4.0. To use the core math library in your own extension, you need to add the npymath compile and link options to your extension in your setup.py:

```python
>>> from numpy.distutils.misc_util import get_info
>>> info = get_info('npymath')
>>> config.add_extension('foo', sources=['foo.c'], extra_info=**info)
```

In other words, the usage of info is exactly the same as when using blas_info and co.
CHAPTER SIX

NUMPY INTERNALS

6.1 Numpy C Code Explanations

Fanaticism consists of redoubling your efforts when you have forgotten your aim. — George Santayana

An authority is a person who can tell you more about something than you really care to know. — Unknown

This Chapter attempts to explain the logic behind some of the new pieces of code. The purpose behind these explanations is to enable somebody to be able to understand the ideas behind the implementation somewhat more easily than just staring at the code. Perhaps in this way, the algorithms can be improved on, borrowed from, and/or optimized.

6.1.1 Memory model

One fundamental aspect of the ndarray is that an array is seen as a “chunk” of memory starting at some location. The interpretation of this memory depends on the stride information. For each dimension in an \( N \)-dimensional array, an integer (stride) dictates how many bytes must be skipped to get to the next element in that dimension. Unless you have a single-segment array, this stride information must be consulted when traversing through an array. It is not difficult to write code that accepts strides, you just have to use (char *) pointers because strides are in units of bytes. Keep in mind also that strides do not have to be unit-multiples of the element size. Also, remember that if the number of dimensions of the array is 0 (sometimes called a rank-0 array), then the strides and dimensions variables are NULL.

Besides the structural information contained in the strides and dimensions members of the PyArrayObject, the flags contain important information about how the data may be accessed. In particular, the NPY_ALIGNED flag is set when the memory is on a suitable boundary according to the data-type array. Even if you have a contiguous chunk of memory, you cannot just assume it is safe to dereference a data-type-specific pointer to an element. Only if the NPY_ALIGNED flag is set is this a safe operation (on some platforms it will work but on others, like Solaris, it will cause a bus error). The NPY_WRITEABLE should also be ensured if you plan on writing to the memory area of the array. It is also possible to obtain a pointer to an unwriteable memory area. Sometimes, writing to the memory area when the NPY_WRITEABLE flag is not set will just be rude. Other times it can cause program crashes (e.g. a data-area that is a read-only memory-mapped file).

6.1.2 Data-type encapsulation

The data-type is an important abstraction of the ndarray. Operations will look to the data-type to provide the key functionality that is needed to operate on the array. This functionality is provided in the list of function pointers pointed to by the ‘f’ member of the PyArray_Descr structure. In this way, the number of data-types can be extended simply by providing a PyArray_Descr structure with suitable function pointers in the ‘f’ member. For built-in types there are some optimizations that by-pass this mechanism, but the point of the data-type abstraction is to allow new data-types to be added.
One of the built-in data-types, the void data-type allows for arbitrary records containing 1 or more fields as elements of the array. A field is simply another data-type object along with an offset into the current record. In order to support arbitrarily nested fields, several recursive implementations of data-type access are implemented for the void type. A common idiom is to cycle through the elements of the dictionary and perform a specific operation based on the data-type object stored at the given offset. These offsets can be arbitrary numbers. Therefore, the possibility of encountering mis-aligned data must be recognized and taken into account if necessary.

### 6.1.3 N-D Iterators

A very common operation in much of NumPy code is the need to iterate over all the elements of a general, strided, N-dimensional array. This operation of a general-purpose N-dimensional loop is abstracted in the notion of an iterator object. To write an N-dimensional loop, you only have to create an iterator object from an ndarray, work with the dataptr member of the iterator object structure and call the macro `PyArray_ITER_NEXT` (it) on the iterator object to move to the next element. The “next” element is always in C-contiguous order. The macro works by first special casing the C-contiguous, 1-d, and 2-d cases which work very simply.

For the general case, the iteration works by keeping track of a list of coordinate counters in the iterator object. At each iteration, the last coordinate counter is increased (starting from 0). If this counter is smaller then one less than the size of the array in that dimension (a pre-computed and stored value), then the counter is increased and the dataptr member is increased by the strides in that dimension and the macro ends. If the end of a dimension is reached, the counter for the last dimension is reset to zero and the dataptr is moved back to the beginning of that dimension by subtracting the strides value times one less than the number of elements in that dimension (this is also re-computed and stored in the backstrides member of the iterator object). In this case, the macro does not end, but a local dimension counter is decremented so that the next-to-last dimension replaces the role that the last dimension played and the previously-described tests are executed again on the next-to-last dimension. In this way, the dataptr is adjusted appropriately for arbitrary striding.

The coordinates member of the `PyArrayIterObject` structure maintains the current N-d counter unless the underlying array is C-contiguous in which case the coordinate counting is by-passed. The index member of the `PyArrayIterObject` keeps track of the current flat index of the iterator. It is updated by the `PyArray_ITER_NEXT` macro.

### 6.1.4 Broadcasting

In Numeric, broadcasting was implemented in several lines of code buried deep in ufuncobject.c. In NumPy, the notion of broadcasting has been abstracted so that it can be performed in multiple places. Broadcasting is handled by the function `PyArray_Broadcast`. This function requires a `PyArrayMultiIterObject` (or something that is a binary equivalent) to be passed in. The `PyArrayMultiIterObject` keeps track of the broadcasted number of dimensions and size in each dimension along with the total size of the broadcasted result. It also keeps track of the number of arrays being broadcast and a pointer to an iterator for each of the arrays being broadcasted.

The `PyArray_Broadcast` function takes the iterators that have already been defined and uses them to determine the broadcast shape in each dimension (to create the iterators at the same time that broadcasting occurs then use the `PyMultiIter_New` function). Then, the iterators are adjusted so that each iterator thinks it is iterating over an array with the broadcasted size. This is done by adjusting the iterators number of dimensions, and the shape in each dimension. This works because the iterator strides are also adjusted. Broadcasting only adjusts (or adds) length-1 dimensions. For these dimensions, the strides variable is simply set to 0 so that the data-pointer for the iterator over that array doesn’t move as the broadcasting operation operates over the extended dimension.

Broadcasting was always implemented in Numeric using 0-valued strides for the extended dimensions. It is done in exactly the same way in NumPy. The big difference is that now the array of strides is kept track of in a `PyArrayIterObject`, the iterators involved in a broadcasted result are kept track of in a `PyArrayMultiIterObject`, and the `PyArray_Broadcast` call implements the broad-casting rules.
6.1.5 Array Scalars

The array scalars offer a hierarchy of Python types that allow a one-to-one correspondence between the data-type stored in an array and the Python-type that is returned when an element is extracted from the array. An exception to this rule was made with object arrays. Object arrays are heterogeneous collections of arbitrary Python objects. When you select an item from an object array, you get back the original Python object (and not an object array scalar which does exist but is rarely used for practical purposes).

The array scalars also offer the same methods and attributes as arrays with the intent that the same code can be used to support arbitrary dimensions (including 0-dimensions). The array scalars are read-only (immutable) with the exception of the void scalar which can also be written to so that record-array field setting works more naturally (a[0]['f1'] = value).

6.1.6 Advanced (“Fancy”) Indexing

The implementation of advanced indexing represents some of the most difficult code to write and explain. In fact, there are two implementations of advanced indexing. The first works only with 1-d arrays and is implemented to handle expressions involving a.flat[obj]. The second is general-purpose that works for arrays of “arbitrary dimension” (up to a fixed maximum). The one-dimensional indexing approaches were implemented in a rather straightforward fashion, and so it is the general-purpose indexing code that will be the focus of this section.

There is a multi-layer approach to indexing because the indexing code can at times return an array scalar and at other times return an array. The functions with “_nice” appended to their name do this special handling while the function without the _nice appendage always return an array (perhaps a 0-dimensional array). Some special-case optimizations (the index being an integer scalar, and the index being a tuple with as many dimensions as the array) are handled in array_subscript_nice function which is what Python calls when presented with the code “a[obj]”.

These optimizations allow fast single-integer indexing, and also ensure that a 0-dimensional array is not created only to be discarded as the array scalar is returned instead. This provides significant speed-up for code that is selecting many scalars out of an array (such as in a loop). However, it is still not faster than simply using a list to store standard Python scalars, because that is optimized by the Python interpreter itself.

After these optimizations, the array_subscript function itself is called. This function first checks for field selection which occurs when a string is passed as the indexing object. Then, 0-d arrays are given special-case consideration. Finally, the code determines whether or not advanced, or fancy, indexing needs to be performed. If fancy indexing is not needed, then standard view-based indexing is performed using code borrowed from Numeric which parses the indexing object and returns the offset into the data-buffer and the dimensions necessary to create a new view of the array. The strides are also changed by multiplying each stride by the step-size requested along the corresponding dimension.

**Fancy-indexing check**

The fancy_indexing_check routine determines whether or not to use standard view-based indexing or new copy-based indexing. If the indexing object is a tuple, then view-based indexing is assumed by default. Only if the tuple contains an array object or a sequence object is fancy-indexing assumed. If the indexing object is an array, then fancy indexing is automatically assumed. If the indexing object is any other kind of sequence, then fancy-indexing is assumed by default. This is over-ridden to simple indexing if the sequence contains any slice, newaxis, or Ellipsis objects, and no arrays or additional sequences are also contained in the sequence. The purpose of this is to allow the construction of “slicing” sequences which is a common technique for building up code that works in arbitrary numbers of dimensions.

**Fancy-indexing implementation**

The concept of indexing was also abstracted using the idea of an iterator. If fancy indexing is performed, then a PyArrayMapIterObject is created. This internal object is not exposed to Python. It is created in order to handle
the fancy-indexing at a high-level. Both get and set fancy-indexing operations are implemented using this object. Fancy indexing is abstracted into three separate operations: (1) creating the PyArrayMapIterObject from the indexing object, (2) binding the PyArrayMapIterObject to the array being indexed, and (3) getting (or setting) the items determined by the indexing object. There is an optimization implemented so that the PyArrayIterObject (which has it's own less complicated fancy-indexing) is used for indexing when possible.

Creating the mapping object

The first step is to convert the indexing objects into a standard form where iterators are created for all of the index array inputs and all Boolean arrays are converted to equivalent integer index arrays (as if nonzero(arr) had been called). Finally, all integer arrays are replaced with the integer 0 in the indexing object and all of the index-array iterators are “broadcast” to the same shape.

Binding the mapping object

When the mapping object is created it does not know which array it will be used with so once the index iterators are constructed during mapping-object creation, the next step is to associate these iterators with a particular ndarray. This process interprets any ellipsis and slice objects so that the index arrays are associated with the appropriate axis (the axis indicated by the iteraxis entry corresponding to the iterator for the integer index array). This information is then used to check the indices to be sure they are within range of the shape of the array being indexed. The presence of ellipsis and/or slice objects implies a sub-space iteration that is accomplished by extracting a sub-space view of the array (using the index object resulting from replacing all the integer index arrays with 0) and storing the information about where this sub-space starts in the mapping object. This is used later during mapping-object iteration to select the correct elements from the underlying array.

Getting (or Setting)

After the mapping object is successfully bound to a particular array, the mapping object contains the shape of the resulting item as well as iterator objects that will walk through the currently-bound array and either get or set its elements as needed. The walk is implemented using the PyArray_MapIterNext function. This function sets the coordinates of an iterator object into the current array to be the next coordinate location indicated by all of the indexing-object iterators while adjusting, if necessary, for the presence of a sub-space. The result of this function is that the dataptr member of the mapping object structure is pointed to the next position in the array that needs to be copied out or set to some value.

When advanced indexing is used to extract an array, an iterator for the new array is constructed and advanced in phase with the mapping object iterator. When advanced indexing is used to place values in an array, a special “broadcasted” iterator is constructed from the object being placed into the array so that it will only work if the values used for setting have a shape that is “broadcastable” to the shape implied by the indexing object.

6.1.7 Universal Functions

Universal functions are callable objects that take N inputs and produce M outputs by wrapping basic 1-d loops that work element-by-element into full easy-to use functions that seamlessly implement broadcasting, type-checking and buffered coercion, and output-argument handling. New universal functions are normally created in C, although there is a mechanism for creating ufuncs from Python functions (frompyfunc). The user must supply a 1-d loop that implements the basic function taking the input scalar values and placing the resulting scalars into the appropriate output slots as explain n implementation.

Setup

Every ufunc calculation involves some overhead related to setting up the calculation. The practical significance of this overhead is that even though the actual calculation of the ufunc is very fast, you will be able to write array and type-specific code that will work faster for small arrays than the ufunc. In particular, using ufuncs to perform many calculations on 0-d arrays will be slower than other Python-based solutions (the silently-imported scalarmath
module exists precisely to give array scalars the look-and-feel of ufunc-based calculations with significantly reduced overhead).

When a ufunc is called, many things must be done. The information collected from these setup operations is stored in a loop-object. This loop object is a C-structure (that could become a Python object but is not initialized as such because it is only used internally). This loop object has the layout needed to be used with PyArray_Broadcast so that the broadcasting can be handled in the same way as it is handled in other sections of code.

The first thing done is to look-up in the thread-specific global dictionary the current values for the buffer-size, the error mask, and the associated error object. The state of the error mask controls what happens when an error-condition is found. It should be noted that checking of the hardware error flags is only performed after each 1-d loop is executed. This means that if the input and output arrays are contiguous and of the correct type so that a single 1-d loop is performed, then the flags may not be checked until all elements of the array have been calculated. Looking up these values in a thread-specific dictionary takes time which is easily ignored for all but very small arrays.

After checking, the thread-specific global variables, the inputs are evaluated to determine how the ufunc should proceed and the input and output arrays are constructed if necessary. Any inputs which are not arrays are converted to arrays (using context if necessary). Which of the inputs are scalars (and therefore converted to 0-d arrays) is noted.

Next, an appropriate 1-d loop is selected from the 1-d loops available to the ufunc based on the input array types. This 1-d loop is selected by trying to match the signature of the data-types of the inputs against the available signatures. The signatures corresponding to built-in types are stored in the types member of the ufunc structure. The signatures corresponding to user-defined types are stored in a linked-list of function-information with the head element stored as a CObject in the userloops dictionary keyed by the data-type number (the first user-defined type in the argument list is used as the key). The signatures are searched until a signature is found to which the input arrays can all be cast safely (ignoring any scalar arguments which are not allowed to determine the type of the result). The implication of this search procedure is that “lesser types” should be placed below “larger types” when the signatures are stored. If no 1-d loop is found, then an error is reported. Otherwise, the argument_list is updated with the stored signature — in case casting is necessary and to fix the output types assumed by the 1-d loop.

If the ufunc has 2 inputs and 1 output and the second input is an Object array then a special-case check is performed so that NotImplemented is returned if the second input is not an ndarray, has the __array_priority__ attribute, and has an __r{op}__ special method. In this way, Python is signaled to give the other object a chance to complete the operation instead of using generic object-array calculations. This allows (for example) sparse matrices to override the multiplication operator 1-d loop.

For input arrays that are smaller than the specified buffer size, copies are made of all non-contiguous, mis-aligned, or out-of-byteorder arrays to ensure that for small arrays, a single-loop is used. Then, array iterators are created for all the input arrays and the resulting collection of iterators is broadcast to a single shape.

The output arguments (if any) are then processed and any missing return arrays are constructed. If any provided output array doesn’t have the correct type (or is mis-aligned) and is smaller than the buffer size, then a new output array is constructed with the special UPDATEIFCOPY flag set so that when it is DECREF’d on completion of the function, it’s contents will be copied back into the output array. Iterators for the output arguments are then processed.

Finally, the decision is made about how to execute the looping mechanism to ensure that all elements of the input arrays are combined to produce the output arrays of the correct type. The options for loop execution are one-loop (for contiguous, aligned, and correct data-type), strided-loop (for non-contiguous but still aligned and correct data-type), and a buffered loop (for mis-aligned or incorrect data-type situations). Depending on which execution method is called for, the loop is then setup and computed.

**Function call**

This section describes how the basic universal function computation loop is setup and executed for each of the three different kinds of execution possibilities. If NPY_ALLOW_THREADS is defined during compilation, then the Python Global Interpreter Lock (GIL) is released prior to calling all of these loops (as long as they don’t involve object arrays).
It is re-acquired if necessary to handle error conditions. The hardware error flags are checked only after the 1-d loop is calculated.

**One Loop**

This is the simplest case of all. The ufunc is executed by calling the underlying 1-d loop exactly once. This is possible only when we have aligned data of the correct type (including byte-order) for both input and output and all arrays have uniform strides (either contiguous, 0-d, or 1-d). In this case, the 1-d computational loop is called once to compute the calculation for the entire array. Note that the hardware error flags are only checked after the entire calculation is complete.

**Strided Loop**

When the input and output arrays are aligned and of the correct type, but the striding is not uniform (non-contiguous and 2-d or larger), then a second looping structure is employed for the calculation. This approach converts all of the iterators for the input and output arguments to iterate over all but the largest dimension. The inner loop is then handled by the underlying 1-d computational loop. The outer loop is a standard iterator loop on the converted iterators. The hardware error flags are checked after each 1-d loop is completed.

**Buffered Loop**

This is the code that handles the situation whenever the input and/or output arrays are either misaligned or of the wrong data-type (including being byte-swapped) from what the underlying 1-d loop expects. The arrays are also assumed to be non-contiguous. The code works very much like the strided loop except for the inner 1-d loop is modified so that pre-processing is performed on the inputs and post-processing is performed on the outputs in bufsize chunks (where bufsize is a user-settable parameter). The underlying 1-d computational loop is called on data that is copied over (if it needs to be). The setup code and the loop code is considerably more complicated in this case because it has to handle:

- memory allocation of the temporary buffers
- deciding whether or not to use buffers on the input and output data (mis-aligned and/or wrong data-type)
- copying and possibly casting data for any inputs or outputs for which buffers are necessary.
- special-casing Object arrays so that reference counts are properly handled when copies and/or casts are necessary.
- breaking up the inner 1-d loop into bufsize chunks (with a possible remainder).

Again, the hardware error flags are checked at the end of each 1-d loop.

**Final output manipulation**

Ufuncs allow other array-like classes to be passed seamlessly through the interface in that inputs of a particular class will induce the outputs to be of that same class. The mechanism by which this works is the following. If any of the inputs are not ndarrays and define the `__array_wrap__` method, then the class with the largest `__array_priority__` attribute determines the type of all the outputs (with the exception of any output arrays passed in). The `__array_wrap__` method of the input array will be called with the ndarray being returned from the ufunc as it’s input. There are two calling styles of the `__array_wrap__` function supported. The first takes the ndarray as the first argument and a tuple of “context” as the second argument. The context is (ufunc, arguments, output argument number). This is the first call tried. If a TypeError occurs, then the function is called with just the ndarray as the first argument.

**Methods**

Their are three methods of ufuncs that require calculation similar to the general-purpose ufuncs. These are reduce, accumulate, and reduceat. Each of these methods requires a setup command followed by a loop. There are four loop
styles possible for the methods corresponding to no-elements, one-element, strided-loop, and buffered-loop. These are the same basic loop styles as implemented for the general purpose function call except for the no-element and one-element cases which are special-cases occurring when the input array objects have 0 and 1 elements respectively.

Setup

The setup function for all three methods is `construct_reduce`. This function creates a reducing loop object and fills it with parameters needed to complete the loop. All of the methods only work on ufuncs that take 2-inputs and return 1 output. Therefore, the underlying 1-d loop is selected assuming a signature of `[otype, otype, otype]` where `otype` is the requested reduction data-type. The buffer size and error handling is then retrieved from (per-thread) global storage. For small arrays that are mis-aligned or have incorrect data-type, a copy is made so that the un-buffered section of code is used. Then, the looping strategy is selected. If there is 1 element or 0 elements in the array, then a simple looping method is selected. If the array is not mis-aligned and has the correct data-type, then strided looping is selected. Otherwise, buffered looping must be performed. Looping parameters are then established, and the return array is constructed. The output array is of a different shape depending on whether the method is reduce, accumulate, or reduceat. If an output array is already provided, then it's shape is checked. If the output array is not C-contiguous, aligned, and of the correct data-type, then a temporary copy is made with the UPDATEIFCOPY flag set. In this way, the methods will be able to work with a well-behaved output array but the result will be copied back into the true output array when the method computation is complete. Finally, iterators are set up to loop over the correct axis (depending on the value of axis provided to the method) and the setup routine returns to the actual computation routine.

Reduce

All of the ufunc methods use the same underlying 1-d computational loops with input and output arguments adjusted so that the appropriate reduction takes place. For example, the key to the functioning of reduce is that the 1-d loop is called with the output and the second input pointing to the same position in memory and both having a step-size of 0. The first input is pointing to the input array with a step-size given by the appropriate stride for the selected axis. In this way, the operation performed is

\[ o = \]
\[ o[0] = i[0] \]
\[ o[k] = i[k] \text{<op>} o[k - 1] k = 1 \ldots N \]

where \( N + 1 \) is the number of elements in the input, \( i \), \( o \) is the output, and \( i[k] \) is the \( k^{th} \) element of \( i \) along the selected axis. This basic operations is repeated for arrays with greater than 1 dimension so that the reduction takes place for every 1-d sub-array along the selected axis. An iterator with the selected dimension removed handles this looping.

For buffered loops, care must be taken to copy and cast data before the loop function is called because the underlying loop expects aligned data of the correct data-type (including byte-order). The buffered loop must handle this copying and casting prior to calling the loop function on chunks no greater than the user-specified bufsize.

Accumulate

The accumulate function is very similar to the reduce function in that the output and the second input both point to the output. The difference is that the second input points to memory one stride behind the current output pointer. Thus, the operation performed is

\[ o[0] = \]
\[ o[k] = i[k] \text{<op>} o[k - 1] k = 1 \ldots N. \]

The output has the same shape as the input and each 1-d loop operates over \( N \) elements when the shape in the selected axis is \( N + 1 \). Again, buffered loops take care to copy and cast the data before calling the underlying 1-d computational loop.

Reduceat

The reduceat function is a generalization of both the reduce and accumulate functions. It implements a reduce over ranges of the input array specified by indices. The extra indices argument is checked to be sure that every input
is not too large for the input array along the selected dimension before the loop calculations take place. The loop implementation is handled using code that is very similar to the reduce code repeated as many times as there are elements in the indices input. In particular: the first input pointer passed to the underlying 1-d computational loop points to the input array at the correct location indicated by the index array. In addition, the output pointer and the second input pointer passed to the underlying 1-d loop point to the same position in memory. The size of the 1-d computational loop is fixed to be the difference between the current index and the next index (when the current index is the last index, then the next index is assumed to be the length of the array along the selected dimension). In this way, the 1-d loop will implement a reduce over the specified indices.

Mis-aligned or a loop data-type that does not match the input and/or output data-type is handled using buffered code where-in data is copied to a temporary buffer and cast to the correct data-type if necessary prior to calling the underlying 1-d function. The temporary buffers are created in (element) sizes no bigger than the user settable buffer-size value. Thus, the loop must be flexible enough to call the underlying 1-d computational loop enough times to complete the total calculation in chunks no bigger than the buffer-size.

6.2 Internal organization of numpy arrays

It helps to understand a bit about how numpy arrays are handled under the covers to help understand numpy better. This section will not go into great detail. Those wishing to understand the full details are referred to Travis Oliphant’s book “Guide to Numpy”.

Numpy arrays consist of two major components, the raw array data (from now on, referred to as the data buffer), and the information about the raw array data. The data buffer is typically what people think of as arrays in C or Fortran, a contiguous (and fixed) block of memory containing fixed sized data items. Numpy also contains a significant set of data that describes how to interpret the data in the data buffer. This extra information contains (among other things):

1. The basic data element’s size in bytes
2. The start of the data within the data buffer (an offset relative to the beginning of the data buffer).
3. The number of dimensions and the size of each dimension
4. The separation between elements for each dimension (the ‘stride’). This does not have to be a multiple of the element size
5. The byte order of the data (which may not be the native byte order)
6. Whether the buffer is read-only
7. Information (via the dtype object) about the interpretation of the basic data element. The basic data element may be as simple as a int or a float, or it may be a compound object (e.g., struct-like), a fixed character field, or Python object pointers.
8. Whether the array is to interpreted as C-order or Fortran-order.

This arrangement allow for very flexible use of arrays. One thing that it allows is simple changes of the metadata to change the interpretation of the array buffer. Changing the byteorder of the array is a simple change involving no rearrangement of the data. The shape of the array can be changed very easily without changing anything in the data buffer or any data copying at all.

Among other things that are made possible is one can create a new array metadata object that uses the same data buffer to create a new view of that data buffer that has a different interpretation of the buffer (e.g., different shape, offset, byte order, strides, etc) but shares the same data bytes. Many operations in numpy do just this such as slices. Other operations, such as transpose, don’t move data elements around in the array, but rather change the information about the shape and strides so that the indexing of the array changes, but the data in the doesn’t move.

Typically these new versions of the array metadata but the same data buffer are new ‘views’ into the data buffer. There is a different ndarray object, but it uses the same data buffer. This is why it is necessary to force copies through use of the .copy() method if one really wants to make a new and independent copy of the data buffer.
New views into arrays mean the object reference counts for the data buffer increase. Simply doing away with the original array object will not remove the data buffer if other views of it still exist.

6.3 Multidimensional Array Indexing Order Issues

What is the right way to index multi-dimensional arrays? Before you jump to conclusions about the one and true way to index multi-dimensional arrays, it pays to understand why this is a confusing issue. This section will try to explain in detail how numpy indexing works and why we adopt the convention we do for images, and when it may be appropriate to adopt other conventions.

The first thing to understand is that there are two conflicting conventions for indexing 2-dimensional arrays. **Matrix notation** uses the first index to indicate which row is being selected and the second index to indicate which column is selected. This is opposite the **geometrically oriented-convention** for images where people generally think the first index represents x position (i.e., column) and the second represents y position (i.e., row). This alone is the source of much confusion; matrix-oriented users and image-oriented users expect two different things with regard to indexing.

The second issue to understand is how indices correspond to the order the array is stored in memory. In Fortran the first index is the most rapidly varying index when moving through the elements of a two-dimensional array as it is stored in memory. If you adopt the matrix convention for indexing, then this means the matrix is stored one column at a time (since the first index moves to the next row as it changes). Thus Fortran is considered a Column-major language. C has just the opposite convention. In C, the last index changes most rapidly as one moves through the array as stored in memory. Thus C is a Row-major language. The matrix is stored by rows. Note that in both cases it presumes that the matrix convention for indexing is being used, i.e., for both Fortran and C, the first index is the row. Note this convention implies that the indexing convention is invariant and that the data order changes to keep that so.

But that’s not the only way to look at it. Suppose one has large two-dimensional arrays (images or matrices) stored in data files. Suppose the data are stored by rows rather than by columns. If we are to preserve our index convention (whether matrix or image) that means that depending on the language we use, we may be forced to reorder the data if it is read into memory to preserve our indexing convention. For example if we read row-ordered data into memory without reordering, it will match the matrix indexing convention for C, but not for Fortran. Conversely, it will match the image indexing convention for Fortran, but not for C. For C, if one is using data stored in row order, and one wants to preserve the image index convention, the data must be reordered when reading into memory.

In the end, which you do for Fortran or C depends on which is more important, not reordering data or preserving the indexing convention. For large images, reordering data is potentially expensive, and often the indexing convention is inverted to avoid that.

The situation with numpy makes this issue yet more complicated. The internal machinery of numpy arrays is flexible enough to accept any ordering of indices. One can simply reorder indices by manipulating the internal stride information for arrays without reordering the data at all. Numpy will know how to map the new index order to the data without moving the data.

So if this is true, why not choose the index order that matches what you most expect? In particular, why not define row-ordered images to use the image convention? (This is sometimes referred to as the Fortran convention vs the C convention, thus the ‘C’ and ‘FORTRAN’ order options for array ordering in numpy.) The drawback of doing this is potential performance penalties. It’s common to access the data sequentially, either implicitly in array operations or explicitly by looping over rows of an image. When that is done, then the data will be accessed in non-optimal order. As the first index is incremented, what is actually happening is that elements spaced far apart in memory are being sequentially accessed, with usually poor memory access speeds. For example, for a two dimensional image ‘im’ defined so that im[0, 10] represents the value at x=0, y=10. To be consistent with usual Python behavior then im[0] would represent a column at x=0. Yet that data would be spread over the whole array since the data are stored in row order. Despite the flexibility of numpy’s indexing, it can’t really paper over the fact basic operations are rendered inefficient because of data order or that getting contiguous subarrays is still awkward (e.g., im[:,0] for the first row, vs im[0]), thus one can’t use an idiom such as for row in im; for col in im does work, but doesn’t yield contiguous column data.
As it turns out, numpy is smart enough when dealing with ufuncs to determine which index is the most rapidly varying one in memory and uses that for the innermost loop. Thus for ufuncs there is no large intrinsic advantage to either approach in most cases. On the other hand, use of .flat with a FORTRAN ordered array will lead to non-optimal memory access as adjacent elements in the flattened array (iterator, actually) are not contiguous in memory.

Indeed, the fact is that Python indexing on lists and other sequences naturally leads to an outside-to inside ordering (the first index gets the largest grouping, the next the next largest, and the last gets the smallest element). Since image data are normally stored by rows, this corresponds to position within rows being the last item indexed.

If you do want to use Fortran ordering realize that there are two approaches to consider: 1) accept that the first index is just not the most rapidly changing in memory and have all your I/O routines reorder your data when going from memory to disk or visa versa, or use numpy’s mechanism for mapping the first index to the most rapidly varying data. We recommend the former if possible. The disadvantage of the latter is that many of numpy’s functions will yield arrays without Fortran ordering unless you are careful to use the ‘order’ keyword. Doing this would be highly inconvenient.

Otherwise we recommend simply learning to reverse the usual order of indices when accessing elements of an array. Granted, it goes against the grain, but it is more in line with Python semantics and the natural order of the data.
Large parts of this manual originate from Travis E. Oliphant's book *Guide to Numpy* (which generously entered Public Domain in August 2008). The reference documentation for many of the functions are written by numerous contributors and developers of Numpy, both prior to and during the Numpy Documentation Marathon.

Please help to improve NumPy’s documentation! Instructions on how to join the ongoing documentation marathon can be found on the scipy.org website.


OpenDocument-formula-20090508.odt


A

absolute() (in module numpy), 618
accumulate
   ufunc methods, 913
accumulate() (numpy.ufunc method), 279
add() (in module numpy), 605
add() (in module numpy.core.defchararray), 813
add_data_dir() (numpy.distutils.misc_util.Configuration method), 837
add_data_files() (numpy.distutils.misc_util.Configuration method), 835
add_extension() (numpy.distutils.misc_util.Configuration method), 838
add_headers() (numpy.distutils.misc_util.Configuration method), 838
add_include_dirs() (numpy.distutils.misc_util.Configuration method), 838
add_installed_library() (numpy.distutils.misc_util.Configuration method), 839
add_library() (numpy.distutils.misc_util.Configuration method), 839
add_npy_pkg_config() (numpy.distutils.misc_util.Configuration method), 840
add_scripts() (numpy.distutils.misc_util.Configuration method), 839
add_subpackage() (numpy.distutils.misc_util.Configuration method), 835
alignment (C member), 851
alignment (numpy.dtype attribute), 55
all() (in module numpy), 520
all() (in module numpy.ma), 150, 682
all() (numpy.ma.MaskedArray method), 121, 158, 690
all() (numpy.ndarray method), 34
all_strings() (in module numpy.distutils.misc_util), 834
allclose() (module numpy), 532
allclose() (module numpy.distutils.misc_util), 834
alterdot() (in module numpy), 798
allequal() (module numpy), 257, 789
alpath() (module numpy.distutils.misc_util), 834
anom() (in module numpy.ma), 216, 748
anom() (module numpy.ma), 122, 228, 760

A

absolute() (in module numpy), 618
accumulate
   ufunc methods, 913
accumulate() (numpy.ufunc method), 279
add() (in module numpy), 605
add() (in module numpy.core.defchararray), 813
add_data_dir() (numpy.distutils.misc_util.Configuration method), 837
add_data_files() (numpy.distutils.misc_util.Configuration method), 835
add_extension() (numpy.distutils.misc_util.Configuration method), 838
add_headers() (numpy.distutils.misc_util.Configuration method), 838
add_include_dirs() (numpy.distutils.misc_util.Configuration method), 838
add_installed_library() (numpy.distutils.misc_util.Configuration method), 839
add_library() (numpy.distutils.misc_util.Configuration method), 839
add_npy_pkg_config() (numpy.distutils.misc_util.Configuration method), 840
add_scripts() (numpy.distutils.misc_util.Configuration method), 839
add_subpackage() (numpy.distutils.misc_util.Configuration method), 835
alignment (C member), 851
alignment (numpy.dtype attribute), 55
all() (in module numpy), 520
all() (in module numpy.ma), 150, 682
all() (numpy.ma.MaskedArray method), 121, 158, 690
all() (numpy.ndarray method), 34
all_strings() (in module numpy.distutils.misc_util), 834
allclose() (module numpy), 532
allclose() (module numpy.distutils.misc_util), 834
alterdot() (in module numpy), 798
allequal() (module numpy), 257, 789
alpath() (module numpy.distutils.misc_util), 834
anom() (in module numpy.ma), 216, 748
anom() (module numpy.ma), 122, 228, 760
anomalies() (in module numy.ma), 217, 749
any() (in module numy), 521
any() (in module numy.ma), 151, 683
any() (numy.ma.MaskedArray method), 122, 159, 691
any() (numy.ndarray method), 34
ao (C member), 857
append() (in module numy), 346
appendpath() (in module numy.distutils.misc_util), 834
apply_along_axis() (in module numy), 625
apply_along_axis() (in module numy.ma), 258, 791
apply_over_axes() (in module numy), 626
arange() (in module numy), 306
arange() (in module numy.ma), 259, 792
arccos() (in module numy), 568
arccosh() (in module numy), 579
arcsin() (in module numy), 568
arcsinh() (in module numy), 570
arctan() (in module numy), 580
arctan2() (in module numy), 582
argmax (C member), 852
argmax() (in module numy), 513
argmax() (in module numy.ma), 235, 767
argmax() (numy.ma.MaskedArray method), 111, 237, 769
argmax() (numy.ndarray method), 31
argmin() (in module numy), 515
argmin() (in module numy.ma), 235, 767
argmin() (numy.ma.MaskedArray method), 112, 237, 769
argmin() (numy.ndarray method), 31
argsort (C member), 853
argsort() (in module numy), 510
argsort() (in module numy.ma), 239, 771
argsort() (numy.ma.MaskedArray method), 112, 242, 774
argsort() (numy.ndarray method), 29
argwhere() (in module numy), 515
arithmetic, 34, 132
around() (in module numy), 581
around() (in module numy.ma), 255, 787
array
  C-API, 865
  interface, 263
  protocol, 263
array iterator, 73, 908
array scalars, 909
array() (in module numy), 292
array() (in module numy.core.defchararray), 69, 305
array() (in module numy.core.records), 303
array() (in module numy.ma), 77, 142, 674
array_equal() (in module numy), 533
array_equiv() (in module numy), 533
array_repr() (in module numy), 400
array_split() (in module numy), 338
array_str() (in module numy), 401
as_array() (in module numy.ctypeslib), 811
as_ctypes() (in module numy.ctypeslib), 811
asanyarray() (in module numy), 295, 329
asanyarray() (in module numy.ma), 80, 196, 728
asarray() (in module numy), 294, 328
asarray() (in module numy.core.defchararray), 305
asarray() (in module numy.ma), 79, 195, 727
ascontiguousarray() (in module numy), 296
asarray() (in module numy.ma), 331
asfortranarray() (in module numy), 331
asmatrix() (in module numy), 63, 297, 330
asscalar() (in module numy), 332
assert_almost_equal() (in module numy.testing), 800
assert_approx_equal() (in module numy.testing), 801
assert_array_almost_equal() (in module numy.testing), 802
assert_array_equal() (in module numy.testing), 803
assert_array_less() (in module numy.testing), 804
assert_equal() (in module numy.testing), 805
assert_raises() (in module numy.testing), 806
assert_string_equal() (in module numy.testing), 806
assert_warns() (in module numy.testing), 806
astype() (numy.ma.MaskedArray method), 103
astype() (numy.ndarray method), 18
atleast_1d() (in module numy), 323
atleast_1d() (in module numy.ma), 168, 700
atleast_2d() (in module numy), 324
atleast_2d() (in module numy.ma), 169, 701
atleast_3d() (in module numy), 325
atleast_3d() (in module numy.ma), 169, 702
attributes
  ufunc, 275
average() (in module numy), 549
average() (in module numy.ma), 217, 749
axis, 30

B
backstrides (C member), 856
bartlett() (in module numy), 656
base, 3
base (C member), 849, 851, 859
base (numy.generic attribute), 43
base (numy.ma.MaskedArray attribute), 96
base (numy.ndarray attribute), 9
base_repr() (in module numy), 407
baseclass (numy.ma.MaskedArray attribute), 96
beta() (in module numy.random), 463
binary_repr() (in module numy), 406, 543
bincount() (in module numy), 563
binomial() (in module numy.random), 463
bitwise_and() (in module numy), 537
bitwise_or() (in module numy), 537
bitwise_xor() (in module numpy), 538
blackman() (in module numpy), 658
blue_text() (in module numpy.distutils.misc_util), 834
bmat() (in module numpy), 64, 317
broadcast (class in numpy), 75, 325
broadcast_arrays() (in module numpy), 326
broadcastable, 269
broadcasting, 269, 908
buffers, 270
byteorder (C member), 850
byteorder (numpy.dtype attribute), 53
bytes() (in module numpy.random), 461
byteswap() (numpy.generic method), 44
byteswap() (numpy.ma.MaskedArray method), 104
byteswap() (numpy.ndarray method), 19

C

C-API
array, 865
darray, 865, 898
ufunc, 898, 902
C-order, 6
c_() (in module numpy), 353
can_cast() (in module numpy), 377
cancastscalarkindto (C member), 853
cancastto (C member), 854
capitalize() (in module numpy.core.defchararray), 814
cast (C member), 852
castdict (C member), 853
casting rules
ufunc, 273
ceil() (in module numpy), 583
center() (in module numpy.core.defchararray), 814
character arrays, 68
chararray (class in numpy), 68
chararray (class in numpy.core.defchararray), 831
check_return (C member), 855
chisquare() (in module numpy.random), 464
cholesky() (in module numpy.linalg), 437
choose() (in module numpy), 367
choose() (in module numpy.ma), 260, 792
choose() (numpy.ma.MaskedArray method), 113
choose() (numpy.ndarray method), 28
clip() (in module numpy), 616
clip() (in module numpy.ma), 255, 787
clip() (numpy.ma.MaskedArray method), 123, 256, 788
clip() (numpy.ndarray method), 32
code generation, 844
column-major, 6
column_stack() (in module numpy), 333
column_stack() (in module numpy.ma), 171, 178, 704, 710
common_fill_value() (in module numpy.ma), 211, 743
common_type() (in module numpy), 378
compare (C member), 852
comparison, 34, 132
compress() (in module numpy), 369
compress() (numpy.ma.MaskedArray method), 113
compress() (numpy.ndarray method), 29
compress_cols() (in module numpy.ma), 205, 737
compress_rowcols() (in module numpy.ma), 206, 738
compressed() (in module numpy.ma), 206, 738
compressed() (numpy.ma.MaskedArray method), 104, 207, 739
concatenate() (in module numpy), 334
concatenate() (in module numpy.ma), 172, 178, 704, 710
cond() (in module numpy.linalg), 448
Configuration (class in numpy.distutils.misc_util), 833, 835
conj() (in module numpy), 614
conj() (numpy.ma.MaskedArray method), 123
conj() (numpy.ndarray method), 32
conjugate() (in module numpy.ma), 218, 750
conjugate() (numpy.ma.MaskedArray method), 123
correlation
from dict, dtype, 51
from dtype, dtype, 48
from list, dtype, 51
from None, dtype, 48
from string, dtype, 49
from tuple, dtype, 50
from type, dtype, 48
cointer class in numpy.lib.user_array), 72
contiguous class, 72
contiguous, 6
cov() (in module numpy), 557
cov() (in module numpy.ma), 120, 145, 677
cov() (numpy.ma.MaskedArray method), 19
cov() (numpy.ndarray method), 19

cov() (in module numpy.ma), 219, 751

cov() (in module numpy.ma), 219, 751

cov() (module numpy), 557

cov() (module numpy.ma), 120, 145, 677

cov() (module numpy.ma), 19

cov() (module numpy.ma), 557

cov() (module numpy), 557

cov() (module numpy.ma), 120, 145, 677

cov() (module numpy.ma), 19

cov() (module numpy.ma), 557

cov() (module numpy), 557

cov() (module numpy.ma), 120, 145, 677

cov() (module numpy.ma), 19

cov() (module numpy.ma), 557

cov() (module numpy), 557

cov() (module numpy.ma), 120, 145, 677

cov() (module numpy.ma), 19

cov() (module numpy.ma), 557

Index
Cumprod() (in module numpy.ma.MaskedArray method), 123, 229, 753, 761
Cumprod() (numpy.ndarray method), 33
Cumsum() (in module numpy), 33
Cumsum() (in module numpy.ma), 220, 752
Cumsum() (numpy.ma.MaskedArray method), 124, 229, 761
Cumsum() (numpy.ndarray method), 33
Cyan_text() (in module numpy.distutils.misc_util), 834
Cyg2win32() (in module numpy.distutils.misc_util), 834
Data (C member), 848, 855, 860
Data (numpy.generic attribute), 43
Data (numpy.ma.MaskedArray attribute), 95, 158, 690
Data (numpy.ndarray attribute), 9
Dataptr (C member), 857
DataSource (class in numpy), 408
Decode() (in module numpy.core.defchararray), 815
Decorate_methods() (in module numpy.testing), 809
Default_fill_value() (in module numpy.ma), 211, 743, 763
Deg2rad() (in module numpy), 575
Degrees() (in module numpy), 573
Delete() (in module numpy), 344
Deprecated() (in module numpy.testing.decorators), 806
Descr (C member), 849, 860
Descr (numpy.dtype attribute), 55
Det() (in module numpy.linalg), 449
Diag() (in module numpy), 313, 370
Diag() (in module numpy.ma), 244, 776
Diag_indices() (in module numpy), 360
Diag_indices_from() (in module numpy), 361
Diagflat() (in module numpy), 314
Diagonal() (in module numpy), 370
Diagonal() (numpy.ma.MaskedArray method), 114
Diagonal() (numpy.ndarray method), 30
Dict_append() (in module numpy.distutils.misc_util), 834
Diff() (in module numpy), 590
Digitize() (in module numpy), 564
Dimensions (C member), 848, 857
Dims_m1 (C member), 856
Dirichlet() (in module numpy.random.mtrand), 465
Distutils, 833
Divide() (in module numpy), 607
Doc (C member), 855
Dot() (in module numpy), 429
Dot() (in module numpy.ma), 244, 777
Dot_join() (in module numpy.distutils.misc_util), 834
Dotfunc (C member), 854
Dsplit() (in module numpy), 338
Dstack() (in module numpy), 335
Dstack() (in module numpy.ma), 173, 179, 705, 711
Dtype, 907
Construction from dict, 51
Construction from dtype, 48
Construction from list, 51
Construction from None, 48
Construction from string, 49
Construction from tuple, 50
Construction from type, 48
Field, 46
Record, 46
Scalar, 45
Sub-array, 46, 50
dtype (class in numpy), 46, 379
dtype (numpy.generic attribute), 43
dtype (numpy.ma.MaskedArray attribute), 98
dtype (numpy.ndarray attribute), 10
dump() (in module numpy.ma), 210, 742
dump() (numpy.ma.MaskedArray method), 121
dump() (numpy.ndarray method), 18
dumpst() (in module numpy.ma), 210, 742
dumpst() (numpy.ma.MaskedArray method), 121
dumpst() (numpy.ndarray method), 18
E
Ediff1d() (in module numpy), 591
Ediff1d() (in module numpy.ma), 261, 793
Eig() (in module numpy.linalg), 441
Eigh() (in module numpy.linalg), 443
Eigvals() (in module numpy.linalg), 444
Eigvalsh() (in module numpy.linalg), 445
Ellipsis, 57
Elsize (C member), 851
Empty() (in module numpy), 287
Empty() (in module numpy.ma), 145, 677
Empty_like() (in module numpy), 288
Empty_like() (in module numpy.ma), 146, 678
Encode() (in module numpy.core.defchararray), 815
Equal() (in module numpy), 535
Equal() (in module numpy.core.defchararray), 823
Error handling, 270
Erystate (class in numpy), 670
Exp() (in module numpy), 595
Exp2() (in module numpy), 597
Expand_dims() (in module numpy), 327
Expand_dims() (in module numpy.ma), 170, 702
Expm1() (in module numpy), 596
Exponential() (in module numpy.random), 466
Extract() (in module numpy), 519
Eye() (in module numpy), 288
Index 931
from dict
dtype construction, 51
from dtype
dtype construction, 48
from list
dtype construction, 51
from None
dtype construction, 48
from string
dtype construction, 49
from tuple
dtype construction, 50
from type
dtype construction, 48
fromarrays() (in module numpy.core.records), 303
frombuffer() (in module numpy), 298
frombuffer() (in module numpy.ma), 143, 675
fromfile() (in module numpy), 298
fromfile() (in module numpy.core.records), 304
fromfunction() (in module numpy), 299
fromfunction() (in module numpy.core.records), 304
fromiter() (in module numpy), 300
frompyfunc() (in module numpy), 300
fromrecords() (in module numpy.core.records), 304
fromregex() (in module numpy), 306
fromstr (C member), 853
fromstring() (in module numpy), 301, 398
fromstring() (in module numpy.core.records), 304
functions (C member), 855
fv() (in module numpy), 643

G
gamma() (in module numpy.random), 467
generate_config_py() (in module numpy.distutils.misc_util), 834
generic (class in numpy), 44
genfromtxt() (in module numpy), 394
geometric() (in module numpy.random), 469
get_build_temp_dir() (numpy.distutils.misc_util.Configuration method), 841
get_cmd() (in module numpy.distutils.misc_util), 834
get_config_cmd() (numpy.distutils.misc_util.Configuration method), 841
get_dependencies() (in module numpy.distutils.misc_util), 834
get_distutils() (numpy.distutils.misc_util.Configuration method), 835
get_ext_source_files() (in module numpy.distutils.misc_util), 835
get_fill_value() (numpy.ma.MaskedArray method), 139, 215, 747
get_info() (in module numpy.distutils.system_info), 842
get_info() (numpy.distutils.misc_util.Configuration method), 842
<table>
<thead>
<tr>
<th>Function</th>
<th>Module Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_numpy_include_dirs()</td>
<td>(in module numpy.distutils.misc_util) 834</td>
</tr>
<tr>
<td>get_printoptions()</td>
<td>(in module numpy) 405</td>
</tr>
<tr>
<td>get_script_files()</td>
<td>(in module numpy.distutils.misc_util) 835</td>
</tr>
<tr>
<td>get_standard_file()</td>
<td>(in module numpy.distutils.system_info) 842</td>
</tr>
<tr>
<td>get_state()</td>
<td>(in module numpy.random) 506</td>
</tr>
<tr>
<td>get_subpackage()</td>
<td>(numpy.distutils.misc_util.Configuration method) 835</td>
</tr>
<tr>
<td>get_version()</td>
<td>(numpy.distutils.misc_util.Configuration method) 842</td>
</tr>
<tr>
<td>getbuffer()</td>
<td>(in module numpy) 798</td>
</tr>
<tr>
<td>getbufsize()</td>
<td>(in module numpy) 799</td>
</tr>
<tr>
<td>getdata()</td>
<td>(in module numpy.ma) 154, 686</td>
</tr>
<tr>
<td>geterr()</td>
<td>(in module numpy) 668</td>
</tr>
<tr>
<td>geterrcall()</td>
<td>(in module numpy) 670</td>
</tr>
<tr>
<td>geterrobj()</td>
<td>(in module numpy) 672</td>
</tr>
<tr>
<td>getfield()</td>
<td>(numpy.ndarray method) 21</td>
</tr>
<tr>
<td>getitem()</td>
<td>(C member) 852</td>
</tr>
<tr>
<td>getmask()</td>
<td>(in module numpy.ma) 153, 185, 685, 717</td>
</tr>
<tr>
<td>getmaskarray()</td>
<td>(in module numpy.ma) 154, 186, 686, 718</td>
</tr>
<tr>
<td>getslice</td>
<td>ndarray special methods, 57</td>
</tr>
<tr>
<td>gradient()</td>
<td>(in module numpy) 591</td>
</tr>
<tr>
<td>greater()</td>
<td>(in module numpy) 534</td>
</tr>
<tr>
<td>greater()</td>
<td>(in module numpy.core.defchararray) 825</td>
</tr>
<tr>
<td>greater_equal()</td>
<td>(in module numpy) 534</td>
</tr>
<tr>
<td>greater_equal()</td>
<td>(in module numpy.core.defchararray) 824</td>
</tr>
<tr>
<td>green_text()</td>
<td>(in module numpy.distutils.misc_util) 834</td>
</tr>
<tr>
<td>histogramdd()</td>
<td>(in module numpy) 562</td>
</tr>
<tr>
<td>hsplit()</td>
<td>(in module numpy) 339</td>
</tr>
<tr>
<td>hstack()</td>
<td>(in module numpy.ma) 174, 706</td>
</tr>
<tr>
<td>histack()</td>
<td>(in module numpy) 336</td>
</tr>
<tr>
<td>histack()</td>
<td>(in module numpy.ma) 174, 180, 706, 712</td>
</tr>
<tr>
<td>hypergeometric()</td>
<td>(in module numpy.random) 174, 806, 712</td>
</tr>
<tr>
<td>hypot()</td>
<td>(in module numpy) 571</td>
</tr>
<tr>
<td>i0()</td>
<td>(in module numpy) 601</td>
</tr>
<tr>
<td>identity()</td>
<td>(C member) 855</td>
</tr>
<tr>
<td>identity()</td>
<td>(numpy.ufunc attribute) 277</td>
</tr>
<tr>
<td>identity()</td>
<td>(in module numpy) 289</td>
</tr>
<tr>
<td>ids()</td>
<td>(in module numpy.ma) 245, 777</td>
</tr>
<tr>
<td>iinfo()</td>
<td>(class in numpy) 382</td>
</tr>
<tr>
<td>imag()</td>
<td>(numpy.generic attribute) 43</td>
</tr>
<tr>
<td>imag()</td>
<td>(numpy.ma.MaskedArray attribute) 101</td>
</tr>
<tr>
<td>imag()</td>
<td>(numpy.ndarray attribute) 11</td>
</tr>
<tr>
<td>ihfft()</td>
<td>(in module numpy.fft) 425</td>
</tr>
<tr>
<td>iiinfo()</td>
<td>(class in numpy) 382</td>
</tr>
<tr>
<td>inner()</td>
<td>(in module numpy) 289</td>
</tr>
<tr>
<td>inner()</td>
<td>(in module numpy.ma) 247, 779</td>
</tr>
<tr>
<td>innerproduct()</td>
<td>(in module numpy.ma) 246, 778</td>
</tr>
<tr>
<td>insert()</td>
<td>(in module numpy) 344</td>
</tr>
<tr>
<td>interp()</td>
<td>(in module numpy) 623</td>
</tr>
<tr>
<td>intersect1d()</td>
<td>(in module numpy) 653</td>
</tr>
<tr>
<td>inv()</td>
<td>(in module numpy.linalg) 454</td>
</tr>
<tr>
<td>invert()</td>
<td>(in module numpy) 539</td>
</tr>
<tr>
<td>ipmt()</td>
<td>(in module numpy) 648</td>
</tr>
<tr>
<td>irfft()</td>
<td>(in module numpy.fft) 419</td>
</tr>
<tr>
<td>irfft2()</td>
<td>(in module numpy.fft) 421</td>
</tr>
<tr>
<td>irfftn()</td>
<td>(in module numpy.fft) 423</td>
</tr>
<tr>
<td>irr()</td>
<td>(in module numpy) 648</td>
</tr>
<tr>
<td>is_local_src_dir()</td>
<td>(in module numpy.distutils.misc_util) 834</td>
</tr>
<tr>
<td>isalpha()</td>
<td>(in module numpy.core.defchararray) 827</td>
</tr>
<tr>
<td>isbuiltin()</td>
<td>(numpy.dtype attribute) 55</td>
</tr>
<tr>
<td>iscomplex()</td>
<td>(in module numpy) 527</td>
</tr>
</tbody>
</table>

**H**

<table>
<thead>
<tr>
<th>Function</th>
<th>Module Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>hamming()</td>
<td>(in module numpy) 660</td>
</tr>
<tr>
<td>hanning()</td>
<td>(in module numpy) 662</td>
</tr>
<tr>
<td>harden_mask()</td>
<td>(in module numpy.ma) 193, 725</td>
</tr>
<tr>
<td>harden_mask()</td>
<td>(numpy.ma.MaskedArray method) 139, 194, 726</td>
</tr>
<tr>
<td>hardmask</td>
<td>(numpy.ma.MaskedArray attribute) 96</td>
</tr>
<tr>
<td>has_cxx_sources()</td>
<td>(in module numpy.distutils.misc_util) 834</td>
</tr>
<tr>
<td>has_f_sources()</td>
<td>(in module numpy.distutils.misc_util) 834</td>
</tr>
<tr>
<td>hasobject</td>
<td>(C member) 850</td>
</tr>
<tr>
<td>hasobject</td>
<td>(numpy.dtype attribute) 54</td>
</tr>
<tr>
<td>have_safe()</td>
<td>(numpy.distutils.misc_util.Configuration method) 841</td>
</tr>
<tr>
<td>have_f90c()</td>
<td>(numpy.distutils.misc_util.Configuration method) 841</td>
</tr>
<tr>
<td>hffft()</td>
<td>(in module numpy.fft) 424</td>
</tr>
<tr>
<td>histogram()</td>
<td>(in module numpy) 558</td>
</tr>
<tr>
<td>histogram2d()</td>
<td>(in module numpy) 560</td>
</tr>
</tbody>
</table>
iscomplexobj() (in module numpy), 527
iscontiguous() (numpy.ma.MaskedArray method), 136
isdecimal() (in module numpy.core.defchararray), 828
isdigit() (in module numpy.core.defchararray), 828
isfinite() (in module numpy), 522
isfortran() (in module numpy), 528
isinf() (in module numpy), 523
isnan() (in module numpy), 524
isnative (numpy.dtype attribute), 55
isneginf() (in module numpy), 525
isnumeric() (in module numpy.core.defchararray), 829
isposinf() (in module numpy), 526
isreal() (in module numpy), 526
isrealobj() (in module numpy), 529
isscalar() (in module numpy), 529
issctype() (in module numpy), 384
isspace() (in module numpy.core.defchararray), 829
issubclass_() (in module numpy), 385
issubdtype() (in module numpy), 385
issubsctype() (in module numpy), 385
istitle() (in module numpy.core.defchararray), 829
isupper() (in module numpy.core.defchararray), 829
item() (numpy.ma.MaskedArray method), 115
item() (numpy.ndarray method), 16
itemset() (numpy.ndarray method), 17
itemsize (C member), 860
itemsize (numpy.dtype attribute), 53
itemsize (numpy.generic attribute), 43
itemsize (numpy.ma.MaskedArray attribute), 99
itemsize (numpy.ndarray attribute), 9
iters (C member), 857
ix_() (in module numpy), 358

J

join() (in module numpy.core.defchararray), 816

K

kaiser() (in module numpy), 664
keyword arguments
  ufunc, 275
  kind (C member), 850
  kind (numpy.dtype attribute), 52
knownfailureif() (in module numpy.testing.decorators), 807
kron() (in module numpy), 436

L

laplace() (in module numpy.random), 473
ldexp() (in module numpy), 604
left_shift() (in module numpy), 540
len (C member), 859
less() (in module numpy), 535
less() (in module numpy.core.defchararray), 825
less_equal() (in module numpy), 535
less_equal() (in module numpy.core.defchararray), 824
lexsort() (in module numpy), 509
LinAlgError, 456
linspace() (in module numpy), 307
listpickle (C member), 854
ljust() (in module numpy.core.defchararray), 816
load() (in module numpy), 389
load() (in module numpy.ma), 210, 742
load_library() (in module numpy.ctypeslib), 811
loads() (in module numpy.ma), 211, 743
loadtxt() (in module numpy), 302, 392
log() (in module numpy), 597
log10() (in module numpy), 598
log1p() (in module numpy), 599
log2() (in module numpy), 599
logaddexp() (in module numpy), 600
logaddexp2() (in module numpy), 600
logical_and() (in module numpy), 530
logical_not() (in module numpy), 531
logical_or() (in module numpy), 530
logical_xor() (in module numpy), 531
logistic() (in module numpy.random), 474
lognormal() (in module numpy.random), 475
logseries() (in module numpy.random), 477
logspace() (in module numpy), 309
lookfor() (in module numpy), 795
lower() (in module numpy.core.defchararray), 816
lstrip() (in module numpy.core.defchararray), 817
linspace() (in module numpy.linalg), 452

M

MachAr (class in numpy), 383
make_config_py() (numpy.distutils.misc_util.Configuration method), 842
make_mask() (in module numpy.ma), 182, 714
make_mask_descr() (in module numpy.ma), 184, 717
make_mask_none() (in module numpy.ma), 183, 715
make_svn_version_py() (numpy.distutils.misc_util.Configuration method), 842
mask (numpy.ma.masked_array attribute), 187, 719
mask (numpy.ma.MaskedArray attribute), 95, 158, 690
mask_cols() (in module numpy.ma), 190, 722
mask_indices() (in module numpy), 361
mask_or() (in module numpy.ma), 184, 191, 716, 723
mask_rowcols() (in module numpy.ma), 191, 723
mask_rows() (in module numpy.ma), 192, 725
masked (in module numpy.ma), 94
masked arrays, 76
masked_all() (in module numpy.ma), 147, 679
masked_all_like() (in module numpy.ma), 147, 679
masked_array (in module numpy.ma), 78, 142, 674
masked_equal() (in module numpy.ma), 81, 197, 729
masked_greater() (in module numpy.ma), 82, 198, 730
masked_greater_equal() (in module numpy.ma), 82, 198, 730
masked_inside() (in module numpy.ma), 82, 198, 730
masked_invalid() (in module numpy.ma), 83, 199, 731
masked_less() (in module numpy.ma), 83, 199, 731
masked_less_equal() (in module numpy.ma), 84, 200, 732
masked_not_equal() (in module numpy.ma), 84, 200, 732
masked_object() (in module numpy.ma), 84, 200, 732
masked_outside() (in module numpy.ma), 85, 201, 733
masked_print_options (in module numpy.ma), 95
masked_values() (in module numpy.ma), 86, 202, 734
masked_where() (in module numpy.ma), 87, 203, 735
MaskedArray (class in numpy.ma), 95
MaskType (in module numpy.ma), 141, 673
mat() (in module numpy), 317
matrix, 62
matrix (class in numpy), 63
matrix_power() (in module numpy.linalg), 435
max() (in module numpy.ma), 235, 767
max() (numpy.ma.MaskedArray method), 125, 238, 770
maximum() (in module numpy), 620
maximum_fill_value() (in module numpy.ma), 212, 213, 744, 745
mean() (in module numpy.ma), 550
mean() (in module numpy.ma), 221, 753
mean() (numpy.ma.MaskedArray method), 125, 229, 761
median() (in module numpy), 551
median() (in module numpy.ma), 222, 754
memmap (class in numpy), 65, 401
memory maps, 65
memory model
ndarray, 907
ndarray (class in numpy), 4
ndenumerate (class in numpy), 74, 375
ndim (numpy.generic attribute), 43
ndim (numpy.ma.MaskedArray attribute), 99
ndim (numpy.ndarray attribute), 8
ndindex (class in numpy), 376
ndpointer() (in module numpy.ctypeslib), 811
negative() (in module numpy), 606
negative_binomial() (in module numpy.random), 811
newaxis, 57
newaxis (in module numpy), 58
newbuffer() (in module numpy), 798
newbyteorder() (numpy.dtype method), 55
nin (C member), 854
nin (numpy.ufunc attribute), 275
NO_IMPORT_ARRAY (C macro), 892
NO_IMPORT_UFUNC (C variable), 902
nomask (in module numpy.ma), 95
non-contiguous, 6
noncentral_chisquare() (in module numpy.random), 481
noncentral_f() (in module numpy.random), 483
nonzero () (in module numpy), 355, 516
nonzero() (in module numpy.ma), 155, 687
nonzero() (numpy.ma.MaskedArray method), 115, 160, 692
nonzero() (numpy.ndarray method), 29
norm() (in module numpy.linalg), 446
normal() (in module numpy.random), 484
not_equal() (in module numpy), 536

N
name (C member), 855
name (numpy.dtype attribute), 53
names (numpy.dtype attribute), 54
nan_to_num() (in module numpy), 622
nanargmax() (in module numpy), 514
nanargmin() (in module numpy), 515
nanmax() (in module numpy), 546
nanmin() (in module numpy), 547
nansum() (in module numpy), 587
nargs (C member), 855
nargs (numpy.ufunc attribute), 276
nbytes (numpy.ma.MaskedArray), 99
nbytes (numpy.ndarray), 9
nd (C member), 848, 857, 860
nd_m1 (C member), 856
ndarray, 61
C-API, 865, 898
memory model, 907
special methods getslice, 57
special methods setslice, 57
view, 58
ndarray (class in numpy), 4
ndenumerate (class in numpy), 74, 375
ndim (numpy.generic attribute), 43
ndim (numpy.ma.MaskedArray attribute), 99
ndim (numpy.ndarray attribute), 8
ndindex (class in numpy), 376
ndpointer() (in module numpy.ctypeslib), 811
negative() (in module numpy), 606
negative_binomial() (in module numpy.random), 480
newaxis, 57
newaxis (in module numpy), 58
newbuffer() (in module numpy), 798
newbyteorder() (numpy.dtype method), 55
nin (C member), 854
nin (numpy.ufunc attribute), 275
NO_IMPORT_ARRAY (C macro), 892
NO_IMPORT_UFUNC (C variable), 902
nomask (in module numpy.ma), 95
non-contiguous, 6
noncentral_chisquare() (in module numpy.random), 481
noncentral_f() (in module numpy.random), 483
nonzero () (in module numpy), 355, 516
nonzero() (in module numpy.ma), 155, 687
nonzero() (numpy.ma.MaskedArray method), 115, 160, 692
nonzero() (numpy.ndarray method), 29
norm() (in module numpy.linalg), 446
normal() (in module numpy.random), 484
not_equal() (in module numpy), 536

Index
not_equal() (in module numpy.core.defchararray), 824
notmasked_contiguous() (in module numpy.ma), 188, 720
notmasked_edges() (in module numpy.ma), 189, 721
nout (C member), 855
nout (numpy.ufunc attribute), 275
nper() (in module numpy), 650
npv() (in module numpy), 645
NPY_1_PI (C variable), 906
NPY_2_PI (C variable), 906
NPY_ALIGNED (C variable), 868, 877
NPY_ALLOW_C_API (C macro), 895
NPY_ALLOW_C_API_DEF (C macro), 895
NPY_BEGIN_ALLOW_THREADS (C macro), 895
NPY_BEGIN_THREADS (C macro), 895
NPY_BEGIN_THREADS_DESCR (C function), 895
NPY_BEHAVED (C variable), 869, 878
NPY_BEHAVED_NS (C variable), 869, 878
NPY_BIG_ENDIAN (C variable), 862
npy_bool (C type), 864
npy_BUFSIZE (C variable), 896
NPY_BYTE_ORDER (C variable), 862
NPY_uint64 (C type), 864
NPY_C_CONTIGUOUS (C variable), 868, 877
NPY_CARRAY (C variable), 869, 878
NPY_CARRAY_RO (C variable), 869, 878
NPY_CLIP (C variable), 882
NPY_CLIPMODE (C type), 897
npy_copysign (C function), 905
NPY_CPU_AMD64 (C variable), 862
NPY_CPU_IA64 (C variable), 862
NPY_CPU_PARISC (C variable), 862
NPY_CPU_PPC (C variable), 862
NPY_CPU_PPC64 (C variable), 862
NPY_CPU_S390 (C variable), 862
NPY_CPU_SPARC (C variable), 862
NPY_CPU_SPARC64 (C variable), 862
NPY_CPU_X86 (C variable), 862
NPY_DEFAULT (C variable), 869, 878
NPY_DISABLE_C_API (C variable), 868, 878
NPY_ELEMENTSTRIDES (C variable), 869
NPY_END_ALLOW_THREADS (C macro), 895
NPY_END_THREADS (C macro), 895
NPY_END_THREADS_DESCR (C function), 895
NPY_ENSUREARRAY (C variable), 868, 878
NPY_ENSURECOPY (C variable), 868, 878
NPY_ENSUREFIELD (C variable), 850
NPY_E (C variable), 905
NPY_EULER (C variable), 906
NPY_F_CONTIGUOUS (C variable), 869, 878
NPY_FAKE (C variable), 868, 877
NPY_FAIL (C variable), 896
NPY_FALSE (C variable), 896
NPY_FARRAY (C variable), 869, 878
NPY_FARRAY_RO (C variable), 869, 878
NPY_FORCECAST (C variable), 868, 878
NPY_FROM_FIELDS (C variable), 850
NPY_INDEX (C variable), 869
NPY_IN_ARRAY (C variable), 869
NPY_IN_FARRAY (C variable), 869
NPY_INFINITY (C variable), 895
NPY_INOUT_ARRAY (C variable), 869
NPY_INOUT_FARRAY (C variable), 869
npy_isfinite (C function), 905
npy_isinf (C function), 905
npy_isnan (C function), 905
NPY_ITEM_HASOBJECT (C variable), 850
NPY_ITEM_IS_POINTER (C variable), 850
NPY_ITEM_LISTPICKLE (C variable), 850
NPY_ITEM_REFCOUNT (C variable), 850
NPY_LITTLE_ENDIAN (C variable), 862
NPY_LOG10E (C variable), 895
NPY_LOG2E (C variable), 895
NPY_LOG10E (C variable), 905
NPY_LOG10E (C variable), 905
NPY_LOGGE10 (C variable), 905
NPY_LOG2E (C variable), 905
NPY_LOG2E (C variable), 905
NPY_MAX_BUFSIZE (C variable), 896
NPY_MAXDIMS (C variable), 896
NPY_MIN_BUFSIZE (C variable), 896
NPY_NAN (C variable), 896
NPY_NEEDS_INIT (C variable), 850
NPY_NEEDS_PYAPI (C variable), 850
npy_nextafter (C function), 906
NPY_NOTSWAPPED (C variable), 869, 878
NPY_NSCALARKINDS (C variable), 905
NPY_NSORTS (C variable), 905
NPY_NUM_FLOATSTYPE (C variable), 896
NPY_NZERO (C variable), 905
NPY_OBJECT_DTYPE_FLAGS (C variable), 850
NPY_ORDER (C type), 897
NPY_OUT_ARRAY (C variable), 869
NPY_OUT_FARRAY (C variable), 869
NPY_OWNDATA (C variable), 877
NPY_PZERO (C variable), 905
NPY_PZERO (C variable), 905
NPY_PZERO (C variable), 905
NPY_RAISE (C variable), 905
NPY_RAISE (C variable), 905
NPY_RAISE (C variable), 905
NPY_SCALAR_PRIORITY (C variable), 897
npy_signbit (C function), 905
NPY_SIZEOF_DOUBLE (C variable), 861
NPY_SIZEOF_FLOAT (C variable), 861
NPY_SIZEOF_INT (C variable), 861
NPY_SIZEOF_LONG (C variable), 861
NPY_SIZEOF_LONG_DOUBLE (C variable), 861
NPY_SIZEOF_LONG_LONG (C variable), 861
NPY_SIZEOF_PYINTPTR_T (C variable), 861
NPY_SIZEOF_PYLONG_LONG (C variable), 861
NPY_SIZEOF_SHORT (C variable), 861
936 Index
NPY_SORTKIND (C type), 897
npy_spacing (C function), 906
NPY_SUBTYPE_PRIORITY (C variable), 896
NPY_SUCCEED (C variable), 896
NPY_TRUE (C variable), 896
NPY_UPDATE_ALL (C variable), 878
NPY_UPDATEIFCOPY (C variable), 869, 877
NPY_USE_GETITEM (C variable), 850
NPY_USE_SETITEM (C variable), 850
NPY_VERSION (C variable), 896
NPY_WRAP (C variable), 882
NPY_WRITEABLE (C variable), 868, 877
ntypes (C member), 855
ntypes (numpy.ufunc attribute), 276
num (numpy.dtype attribute), 52
numiter (C member), 857
numpy (module), 1
numpy.distutils (module), 833
numpy.distutils.misc_util (module), 833
numpy.doc.internals (module), 914
numpy.lib.scimath (module), 810
numpy.fft (module), 408
numpy.lib.scimath (module), 810
numpy.oldnumeric (module), 811
numpy.ndarray (module), 33
numpy.oldnumeric (module), 811
numpy.oldnumeric (module), 811
obj (C member), 855
obj2scctype() (in module numpy), 378
offset, 6
ogrid() (in module numpy), 312, 359
ones() (in module numpy), 290
ones() (in module numpy.ma), 148, 680
ones_like() (in module numpy), 290
operation, 34, 132
operator, 34, 132
outer() (in module numpy), 432
outer() (in module numpy.ma), 247, 780
outer() (numpy.ufunc method), 282
outerproduct() (in module numpy.ma), 249, 781
packbits() (in module numpy), 542
pareto() (in module numpy.random), 485
partition() (in module numpy.core.defchararray), 817
path() (numpy.distutils.misc_util.Configuration method), 841
permutation() (in module numpy.random), 461
piecewise() (in module numpy), 628
pinv() (in module numpy.linalg), 454
place() (in module numpy), 372
ppmt() (in module numpy), 647
prod() (in module numpy), 584
prod() (in module numpy.ma), 223, 755
prod() (numpy.ma.MaskedArray method), 126, 230, 762
prod() (numpy.ndarray method), 33
product() (numpy.ma.MaskedArray method), 127
protocol
array, 263
ptp() (in module numpy), 548
pp() (in module numpy), 236, 768
pp() (numpy.ma.MaskedArray method), 128, 239, 771
pp() (numpy.ndarray method), 32
ptr (C member), 855, 859
put() (in module numpy), 373
put() (numpy.ma.MaskedArray method), 117
put() (numpy.ndarray method), 28
putmask() (in module numpy), 374
pv() (in module numpy), 644
PY_ARRAY_UNIQUE_SYMBOL (C macro), 892
PY_UFUNC_UNIQUE_SYMBOL (C variable), 902
PyArray_All (C function), 884
PyArray_Any (C function), 884
PyArray_ArgMax (C function), 883
PyArray_ArgMin (C function), 883
PyArray_ArgSort (C function), 882
PyArray_ArgType (C function), 875
PyArray_ArrFuncs (C type), 851
PyArray_AsCArray (C function), 884
PyArray_AxisConverter (C function), 891
PyArray_BASE (C function), 866
PyArray_BoolConverter (C function), 891
PyArray_Broadcast (C function), 887
PyArray_BroadcastToShape (C function), 886
PyArray_BufferConverter (C function), 891
PyArray_BYTE (C function), 865
PyArray_Bytestr (C function), 880
PyArray_CanCastSafely (C function), 875
PyArray_CanCastTo (C function), 875
PyArray_CanCoerceScalar (C function), 889
PyArray_Cast (C function), 875
PyArray_CastScalarToType (C function), 889
PyArray_CastTo (C function), 875
PyArray_CastToType (C function), 875
PyArray_EQ (C function), 897
PyArray_GE (C function), 897
PyArray_GT (C function), 897
PyArray_Check (C function), 871
PyArray_CheckAxis (C function), 871
PyArray_CheckExact (C function), 871
PyArray_CheckFromAny (C function), 869
PyArray_CheckScalar (C function), 872
PyArray_CheckStrides (C function), 885
PyArray_CHKFLAGS (C function), 878
PyArray_Choose (C function), 882
PyArray_Chunk (C type), 859
PyArray_CLE (C function), 897
PyArray_Clip (C function), 883
PyArray_CLT (C function), 897
PyArray_CNE (C function), 897
PyArray_CompareLists (C function), 886
PyArray_Compress (C function), 883
PyArray_Concatenate (C function), 885
PyArray_Conjugate (C function), 883
PyArray_ContiguousFromAny (C function), 870
PyArray_Converter (C function), 891
PyArray CONVERTToCommonType (C function), 875
PyArray_Correlate (C function), 885
PyArray_Correlate2 (C function), 885
PyArray_CumProd (C function), 884
PyArray_CumSum (C function), 884
PyArray_DATA (C function), 865
PyArray_DESCR (C function), 866
PyArray_DESCRIPTOR (C function), 871
PyArray CompareLists (C function), 886
PyArray_Compress (C function), 883
PyArray_concatenate (C function), 885
PyArray_Conjugate (C function), 883
PyArray_contiguousFromAny (C function), 870
PyArray_Converter (C function), 891
PyArray_convertToCommonType (C function), 875
PyArray_CopyAndTranspose (C function), 885
PyArray_CopyInto (C function), 871
PyArray_Correlate (C function), 885
PyArray_correlate2 (C function), 885
PyArray_CumProd (C function), 884
PyArray_CumSum (C function), 884
PyArray_DATA (C function), 865
PyArray_DESCR (C function), 866
PyArray_Descr (C type), 849
PyArray_DescrAlignConverter (C function), 890
PyArray_DescrAlignConverter2 (C function), 890
PyArray_DescrConverter (C function), 890
PyArray_DescrConverter2 (C function), 890
PyArray_DescrFromObject (C function), 890
PyArray_DescrFromScalar (C function), 890
PyArray_DescrFromType (C function), 890
PyArray_DescrNew (C function), 890
PyArray_DescrNewByteorder (C function), 890
PyArray_DescrNewFromType (C function), 890
PyArray_Diagonal (C function), 882
PyArray_DIM (C function), 865
PyArray_DIMS (C function), 865
PyArray_Dims (C type), 858
PyArray_Dump (C function), 880
PyArray_Dumps (C function), 880
PyArray_EMPTY (C function), 868
PyArray_empty (C function), 867
PyArray_EnsureArray (C function), 870
PyArray_EquivArrTypes (C function), 874
PyArray_EquivByteorders (C function), 874
PyArray_EquivTypenums (C function), 874
PyArray_EquivTypes (C function), 874
PyArray_FieldNames (C function), 891
PyArray_FillObjectArray (C function), 877
PyArray_FILLWBYTE (C function), 867
PyArray_FillWithScalar (C function), 880
PyArray_FLAGS (C function), 866
PyArray_Flatten (C function), 881
PyArray_Free (C function), 885
PyArray_free (C function), 894
PyArray_FROM_O (C function), 871
PyArray_FROM_OF (C function), 871
PyArray_FROM_OT (C function), 871
PyArray_FROM_OTF (C function), 871
PyArray_FROMANY (C function), 871
PyArray_FillObjectArray (C function), 870
PyArray_FillWithScalar (C function), 880
PyArray_FROMArray (C function), 870
PyArray_FromArray (C function), 870
PyArray_FromArrayAttr (C function), 870
PyArray_FromBuffer (C function), 870
PyArray_FromFile (C function), 870
PyArray_FromInterface (C function), 870
PyArray_FromObject (C function), 870
PyArray_FromScalar (C function), 889
PyArray_FromString (C function), 870
PyArray_FromStructInterface (C function), 870
PyArray_GetCastFunc (C function), 875
PyArray_GETCONTIGUOUS (C function), 871
PyArray_GetEndianness (C function), 862
PyArray_GetField (C function), 879
PyArray_GETITEM (C function), 866
PyArray_HasArrayInterface (C function), 872
PyArray_HasArrayInterfaceType (C function), 872
PyArray_HASFIELDS (C function), 874
PyArray_INCREF (C function), 876
PyArray_InitArrFuncs (C function), 876
PyArray_InnerProduct (C function), 885
PyArray_INTPT1 (C function), 866
PyArray_INTPT2 (C function), 866
PyArray_INTPT3 (C function), 866
PyArray_INTPT4 (C function), 866
PyArray_HasArrayInterface (C function), 872
PyArray_HasArrayInterfaceType (C function), 872
PyArray_HASFIELDS (C function), 874
PyArray_INCREF (C function), 876
PyArray_InitArrFuncs (C function), 866
PyArray_InnerProduct (C function), 885
PyArray_IntpConverter (C function), 892
PyArray_IntpFromSequence (C function), 892
PyArray_ISALIGNED (C function), 879
PyArray_IsAnyScalar (C function), 872
PyArray_ISBEHAVED (C function), 879
PyArray_ISBEHAVED_RO (C function), 879
PyArray_ISBOOL (C function), 874
PyArray_ISBYTESWAPPED (C function), 874
PyArray_ISCARRAY (C function), 879
PyArray_ISCARRAY_RO (C function), 879
PyArray_ISCOMPLEX (C function), 873
PyArray_ISCONTIGUOUS (C function), 879
PyArray_ISEXTENDED (C function), 874
PyArray_ISFARRAY (C function), 879
PyArray_ISFARRAY_RO (C function), 879
PyArray_ISFLEXIBLE (C function), 873
PyArray_ISFLOAT (C function), 873
PyArray_ISFORTRAN (C function), 879
PyArray_ISINTEGER (C function), 872
PyArray_ISNOTSWAPPED (C function), 874
PyArray_ISNUMBER (C function), 873
PyArray_ISOBJECT (C function), 874
PyArray_ISONESEGMENT (C function), 879
PyArray_ISPYTHON (C function), 873
PyArray_IsPythonScalar (C function), 872
PyArray_IsScalar (C function), 872
PyArray_ISSIGNED (C function), 872
PyArray_ISSINGLE (C function), 873
PyArray_ISNUMBER (C function), 872
PyArray_ISOBJECT (C function), 874
PyArray_ISSTRING (C function), 879
PyArray_IsZeroDim (C function), 872
PyArray_Item_INCREF (C function), 876
PyArray_Item_XDECREF (C function), 876
PyArray_ITEMSIZE (C function), 866
PyArray_ITER_DATA (C function), 886
PyArray_ITER_GOTO (C function), 886
PyArray_ITER_GOTO1D (C function), 886
PyArray_ITER_NEXT (C function), 886
PyArray_ITER_RESET (C function), 886
PyArray_ITERAllButAxis (C function), 886
PyArray_IterNew (C function), 887
PyArray_MultiplyIntList (C function), 886
PyArray_MultiplyList (C function), 886
PyArray_MultiplyIntList (C function), 886
PyArray_NBYTES (C function), 866
PyArray_NeighborhoodIterNew (C function), 887
PyArray_New (C function), 867
PyArray_NewCopy (C function), 880
PyArray_NewFromDescr (C function), 867
PyArray_Newshape (C function), 880
PyArray_Nonzero (C function), 882
PyArray(ObjectType (C function), 875
PyArray_One (C function), 876
PyArray_OutputConverter (C function), 891
PyArray_Produc (C function), 884
PyArray_Ptp (C function), 883
PyArray_PutMask (C function), 882
PyArray_PutTo (C function), 881
PyArray_PutTo (C function), 889
PyArray_PyIntAsInt (C function), 875
PyArray_PyIntAsIntp (C function), 876
PyArray_Ravel (C function), 881
PyArray_realloc (C function), 894
PyArray_REFCOUNT (C function), 897
PyArray_RegisterCanCast (C function), 876
PyArray_RegisterCastFunc (C function), 876
PyArray_RegisterDataType (C function), 876
PyArray_RemoveSmallest (C function), 887
PyArray_Repe (C function), 882
PyArray_reshape (C function), 881
PyArray_Resize (C function), 881
PyArray_Return (C function), 889
PyArray_Round (C function), 883
PyArray_SAME (C function), 896
PyArray_Scalar (C function), 889
PyArray_ScalarAsCtype (C function), 889
PyArray_ScalarKind (C function), 889
PyArray_SearchsideConverter (C function), 892
PyArray_SearchSorted (C function), 882
PyArray_SetField (C function), 879
PyArray_SETITEM (C function), 866
PyArray_Std (C function), 883
PyArray_STRIDE (C function), 865
PyArray_STRIDES (C function), 865
PyArray_Sum (C function), 884
PyArray_MultiIter_RESET (C function), 887
PyArray_MultiIterNew (C function), 887
PyArray_MultiIterReset (C function), 886
PyArray_MultiIterList (C function), 886
PyArray_MultiIterList (C function), 886
PyArray_NBYTES (C function), 866
PyArray_NeighborhoodIterNew (C function), 887
PyArray_New (C function), 867
PyArray_NewCopy (C function), 880
PyArray_NewFromDescr (C function), 867
PyArray_Newshape (C function), 880
PyArray_Nonzero (C function), 882
PyArray_ObjectType (C function), 875
PyArray_One (C function), 876
PyArray_OutputConverter (C function), 891
PyArray_Produc (C function), 884
PyArray_Ptp (C function), 883
PyArray_PutMask (C function), 882
PyArray_PutTo (C function), 881
PyArray_PyIntAsInt (C function), 892
PyArray_PyIntAsIntp (C function), 892
PyArray_Ravel (C function), 881
PyArray_realloc (C function), 894
PyArray_REFCOUNT (C function), 897
PyArray_RegisterCanCast (C function), 876
PyArray_RegisterCastFunc (C function), 876
PyArray_RegisterDataType (C function), 876
PyArray_RemoveSmallest (C function), 887
PyArray_Repe (C function), 882
PyArray_reshape (C function), 881
PyArray_Resize (C function), 881
PyArray_Return (C function), 889
PyArray_Round (C function), 883
PyArray_SAME (C function), 896
PyArray_Scalar (C function), 889
PyArray_ScalarAsCtype (C function), 889
PyArray_ScalarKind (C function), 889
PyArray_SearchsideConverter (C function), 892
PyArray_SearchSorted (C function), 882
PyArray_SetField (C function), 879
PyArray_SETITEM (C function), 866
PyArray_Sum (C function), 884
<table>
<thead>
<tr>
<th>Function/Attribute Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyArray_SwapAxes (C function)</td>
<td>881</td>
</tr>
<tr>
<td>PyArray_TakeFrom (C function)</td>
<td>881</td>
</tr>
<tr>
<td>PyArray_ToFile (C function)</td>
<td>880</td>
</tr>
<tr>
<td>PyArray_ToList (C function)</td>
<td>880</td>
</tr>
<tr>
<td>PyArray_ToScalar (C function)</td>
<td>889</td>
</tr>
<tr>
<td>PyArray_ToString (C function)</td>
<td>880</td>
</tr>
<tr>
<td>PyArray_Trace (C function)</td>
<td>883</td>
</tr>
<tr>
<td>PyArray_Transpose (C function)</td>
<td>881</td>
</tr>
<tr>
<td>PyArray_TYPE (C function)</td>
<td>866</td>
</tr>
<tr>
<td>PyArray_Type (C variable)</td>
<td>848</td>
</tr>
<tr>
<td>PyArray_TypeObjectFromType (C function)</td>
<td>889</td>
</tr>
<tr>
<td>PyArray_TypestrConvert (C function)</td>
<td>892</td>
</tr>
<tr>
<td>PyArray_UpdateFlags (C function)</td>
<td>879</td>
</tr>
<tr>
<td>PyArray_ValidType (C function)</td>
<td>876</td>
</tr>
<tr>
<td>PyArray_View (C function)</td>
<td>880</td>
</tr>
<tr>
<td>PyArray_Where (C function)</td>
<td>885</td>
</tr>
<tr>
<td>PyArray_XDECREF (C function)</td>
<td>876</td>
</tr>
<tr>
<td>PyArray_XDECREF_ERR (C function)</td>
<td>897</td>
</tr>
<tr>
<td>PyArray_Zero (C function)</td>
<td>876</td>
</tr>
<tr>
<td>PyArray_ZEROS (C function)</td>
<td>867</td>
</tr>
<tr>
<td>PyArray_Zeros (C function)</td>
<td>867</td>
</tr>
<tr>
<td>PyArrayDescr_Check (C function)</td>
<td>890</td>
</tr>
<tr>
<td>PyArrayDescr_Type (C variable)</td>
<td>849</td>
</tr>
<tr>
<td>PyArrayFlags_Type (C variable)</td>
<td>858</td>
</tr>
<tr>
<td>PyArrayInterface (C type)</td>
<td>859</td>
</tr>
<tr>
<td>PyArrayIter_Check (C function)</td>
<td>886</td>
</tr>
<tr>
<td>PyArrayIter_Type (C variable)</td>
<td>856</td>
</tr>
<tr>
<td>PyArrayIterObject (C type)</td>
<td>856</td>
</tr>
<tr>
<td>PyArrayNeighborhoodIter_Type (C variable)</td>
<td>861</td>
</tr>
<tr>
<td>PyArrayMultiIter_Type (C variable)</td>
<td>857</td>
</tr>
<tr>
<td>PyArrayMultiIterObject (C type)</td>
<td>857</td>
</tr>
<tr>
<td>PyArrayNeighborhhoodIter_Next (C function)</td>
<td>888</td>
</tr>
<tr>
<td>PyArrayNeighborhhoodIter_Reset (C function)</td>
<td>888</td>
</tr>
<tr>
<td>PyArrayNeighborhhoodIter_Type (C variable)</td>
<td>858</td>
</tr>
<tr>
<td>PyArrayNeighborhhoodIterObject (C type)</td>
<td>858</td>
</tr>
<tr>
<td>PyArrayObject (C type)</td>
<td>848</td>
</tr>
<tr>
<td>PyDataMem_FREE (C function)</td>
<td>894</td>
</tr>
<tr>
<td>PyDataMem_NEW (C function)</td>
<td>894</td>
</tr>
<tr>
<td>PyDataMem_RENEW (C function)</td>
<td>894</td>
</tr>
<tr>
<td>PyDataType_FLAGCHK (C function)</td>
<td>850</td>
</tr>
<tr>
<td>PyDataType_HASFIELDS (C function)</td>
<td>874</td>
</tr>
<tr>
<td>PyDataType_ISSBOOL (C function)</td>
<td>874</td>
</tr>
<tr>
<td>PyDataType_ISCOMPLEX (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyDataType_ISEXTENDED (C function)</td>
<td>874</td>
</tr>
<tr>
<td>PyDataType_ISFLEXIBLE (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyDataType_ISFLOAT (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyDataType_ISINTEGER (C function)</td>
<td>872</td>
</tr>
<tr>
<td>PyDataType_ISNUMBER (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyDataType_ISOBJECT (C function)</td>
<td>874</td>
</tr>
<tr>
<td>PyDataType_ISPYTHON (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyDataType_ISSIGNED (C function)</td>
<td>872</td>
</tr>
<tr>
<td>PyDataType_ISSTRING (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyDataType_ISUNSIGNED (C function)</td>
<td>872</td>
</tr>
<tr>
<td>PyDataType_ISUSERDEF (C function)</td>
<td>873</td>
</tr>
<tr>
<td>PyUFunc_checkfperr (C function)</td>
<td>909</td>
</tr>
<tr>
<td>PyUFunc_clearperr (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_d_D (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_d_d (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_DD_D (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_dd_d (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_F_F (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_f_f (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_F_F_As_D_D (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_f_f_As_d_d (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_FF_F (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_ff_f (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_FF_F_As_DD_D (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_ff_f_As_dd_d (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_O_O (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_O_O_method (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc ОО_O (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc ОО_O_method (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_GenericFunction (C function)</td>
<td>899</td>
</tr>
<tr>
<td>PyUFunc_GetPyValues (C function)</td>
<td>900</td>
</tr>
<tr>
<td>PyUFunc_G_G (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_g_g (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_loop1d (C type)</td>
<td>861</td>
</tr>
<tr>
<td>PyUFunc_O_O_method (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc ОО_O (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc ОО_O_method (C function)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_PyFuncData (C type)</td>
<td>901</td>
</tr>
<tr>
<td>PyUFunc_RegisterLoopForType (C function)</td>
<td>899</td>
</tr>
<tr>
<td>PyUFunc_ReplaceloopBySignature (C function)</td>
<td>899</td>
</tr>
<tr>
<td>PyUFunc_Type (C variable)</td>
<td>854</td>
</tr>
<tr>
<td>PyUFuncLoopObject (C type)</td>
<td>860</td>
</tr>
<tr>
<td>PyUFuncObject (C type)</td>
<td>854</td>
</tr>
<tr>
<td>PyUFuncReduceObject (C type)</td>
<td>861</td>
</tr>
</tbody>
</table>
Q

qr() (in module numpy.linalg), 438

R

r_() (in module numpy), 353
rad2deg() (in module numpy), 575
radians() (in module numpy), 573
rand() (in module numpy.random), 457
randint() (in module numpy.random), 458
randn() (in module numpy.random), 457
random_integers() (in module numpy.random), 459
random_sample() (in module numpy.random), 460
RandomState (class in numpy.random.mtrand), 505
RankWarning, 642
rate() (in module numpy), 650
ravel() (in module numpy), 319
ravel() (in module numpy.ma), 163, 695
ravel() (numpy.ma.MaskedArray method), 108, 165, 697
ravel() (numpy.ndarray method), 27
rayleigh() (in module numpy.random), 491
real (numpy.generic attribute), 43
real (numpy.ma.MaskedArray attribute), 101
real (numpy.ndarray attribute), 11
real() (in module numpy), 613
real_if_close() (in module numpy), 622
recarray (class in numpy), 70
reciprocal() (in module numpy), 605
dtype, 46
record (class in numpy), 72
recordmask (numpy.ma.MaskedArray attribute), 95, 158, 690
red_text() (in module numpy.distutils.misc_util), 834
reduce
    ufunc methods, 913
reduce() (numpy.ufunc method), 278
reduceat
    ufunc methods, 913
reduceat() (numpy.ufunc method), 280
remainder() (in module numpy), 612
repeat() (in module numpy), 343
repeat() (numpy.ma.MaskedArray method), 118
repeat() (numpy.ndarray method), 28
replace() (in module numpy.core.defchararray), 818
require() (in module numpy), 332
reshape() (in module numpy), 318, 350
reshape() (in module numpy.ma), 163, 695
reshape() (numpy.ma.MaskedArray method), 109, 165, 697
reshape() (numpy.ndarray method), 23
resize() (in module numpy), 346
resize() (in module numpy.ma), 163, 695
resize() (numpy.ma.MaskedArray method), 109, 166, 698
resize() (numpy.ndarray method), 24
restoredot() (in module numpy), 799
rfft() (in module numpy.fft), 418
rfft2() (in module numpy.fft), 420
rfftn() (in module numpy.fft), 421
rfind() (in module numpy.core.defchararray), 830
right_shift() (in module numpy), 541
rindex() (in module numpy.core.defchararray), 830
rint() (in module numpy), 582
rjost() (in module numpy.core.defchararray), 818
roll() (in module numpy), 351
rollaxis() (in module numpy), 321
roots() (in module numpy), 634
rot90() (in module numpy), 352
round() (in module numpy.ma), 256, 788
round() (numpy.ma.MaskedArray method), 128, 256, 788
round() (numpy.ndarray method), 32
round() (in module numpy), 582
row-major, 6
row_stack() (in module numpy.ma), 176, 708
rpartition() (in module numpy.core.defchararray), 819
rsplit() (in module numpy.core.defchararray), 819
rstrip() (in module numpy.core.defchararray), 819
run_module_suite() (in module numpy.testing), 809
rundocs() (in module numpy.testing), 809
S

s_() (in module numpy), 355
save() (in module numpy), 390
savetxt() (in module numpy), 393
savez() (in module numpy), 390

scalar
dtype, 45
scallarkind (C member), 853
scanfunc (C member), 853
settype2char() (in module numpy), 387
searchsorted() (in module numpy), 518
searchsorted() (numpy.ma.MaskedArray method), 118
searchsorted() (numpy.ndarray method), 29
seed() (in module numpy.random), 505
select() (in module numpy), 372
set_fill_value() (in module numpy), 324, 746
set_fill_value() (numpy.ma.MaskedArray method), 140, 215, 747

set_printoptions() (in module numpy), 404
set_state() (in module numpy.random), 506
set_string_function() (in module numpy), 405
set_verbosity() (in module numpy.distutils.log), 843
setastest() (in module numpy.testing.decorators), 807
setbufsize() (in module numpy), 270, 799
setdiff1d() (in module numpy), 654
seterr() (in module numpy), 270, 666
seterrcall() (in module numpy), 272, 668
seterrobj() (in module numpy), 671
setflags() (numpy.generic method), 45

Index 941
Index