NumPy Reference

Release 1.4.dev

Written by the NumPy community

March 24, 2009

CONTENTS

Release
1.4.dev

Date
March 24, 2009

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*).

CHAPTER

ONE

ARRAY OBJECTS

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.

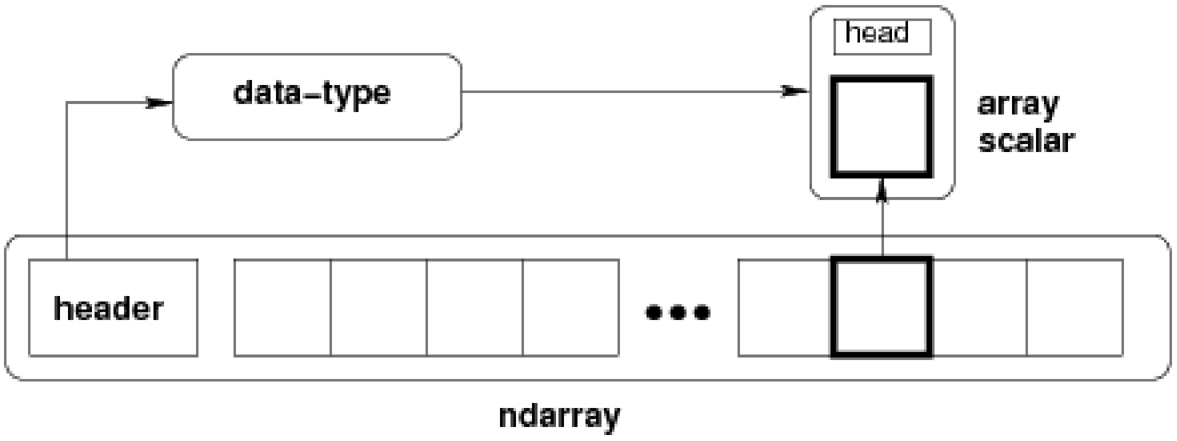

Figure 1.1: **Figure** Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type object that describes the layout of a single fixed-size element of the array, 3) the array-scalar Python object that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

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<type 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

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

```
>>> x[1,2]
6
```

For example *slicing* can produce views of the array:

```
>>> y = x[:,1]
>>> y[0] = 9
>>> x
array([[1, 9, 3],
       [4, 5, 6]])
```

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:

ndarray(shape[,dtype,buffer,offset,...])	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 or a floating point number, etc.).

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 or a floating point number, etc.).

Arrays should be constructed using *array*, *zeros* or *empty* (refer to the `See Also` section below). The parameters given here describe a low-level method for instantiating an array (*ndarray(...)*).

For more information, refer to the *numpy* module and examine the the methods and attributes of an array.

Parameters

shape : tuple of ints

Shape of created array.

dtype : data type, optional

Any object that can be interpreted 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.

Attributes

T : ndarray

Transponent of the array.

data : buffer

Array data in memory.

dtype : data type

Data type, describing the format of the elements in the array.

flags : dict

Dictionary containing information related to memory use, e.g., 'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', and others.

flat : ndarray

Return flattened version of the array as an iterator. The iterator allows assignments, e.g., `x.flat = 3`.

imag : ndarray

Imaginary part of the array.

real : ndarray

Real part of the array.

size : int

Number of elements in the array.

itemsize : int

The size of each element in memory (in bytes).

nbytes : int

The total number of bytes required to store the array data, i.e., `itemsize * size`.

shape : tuple of ints

Shape of the array.

strides : tuple of ints

The step-size required to move from one element to the next in memory. For example, a contiguous `(3, 4)` array of type `int16` in C-order has strides `(8, 2)`. This implies that to move from element to element in memory requires jumps of 2 bytes. To move from row-to-row, one needs to jump 6 bytes at a time (`2 * 4`).

ctypes : ctypes object

Class containing properties of the array needed for interaction with ctypes.

base : ndarray

If the array is a view on another array, that array is its *base* (unless that array is also a view). The *base* array is where the array data is ultimately stored.

See Also:

`array`
Construct an array.

`zeros`
Create an array and fill its allocated memory with zeros.

empty
Create an array, but leave its allocated memory unchanged.

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 exporting 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 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 of arranging the items of an *N*-dimensional array to 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) are special cases of the strided scheme, and correspond to the strides:

$$s_k^{\mathrm{column}} = \prod_{j=0}^{k-1} d_j , \quad s_k^{\mathrm{row}} = \prod_{j=k+1}^{N-1} d_j .$$

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:

`ndarray.flags`	Information about the memory layout of the array.
`ndarray.shape`	Tuple of array dimensions.
`ndarray.strides`	Tuple of bytes to step in each dimension.
`ndarray.ndim`	Number of array dimensions.
`ndarray.data`	Buffer object pointing to the start of the data.
`ndarray.size`	Number of elements in the array.
`ndarray.itemsize`	Length of one element in bytes.
`ndarray.nbytes`	Number of bytes in the array.
`ndarray.base`	Base object if memory is from some other object.

`flags`

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C) :

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F) :

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O) :
The array owns the memory it uses or borrows it from another object.

WRITEABLE (W) :
The data area can be written to.

ALIGNED (A) :
The data and strides are aligned appropriately for the hardware.

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

FNC :
F_CONTIGUOUS and not C_CONTIGUOUS.

FORC :
F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B) :
ALIGNED and WRITEABLE.

CARRAY (CA) :
BEHAVED and C_CONTIGUOUS.

FARRAY (FA) :
BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be also accessed dictionary-like, and using lowercased attribute names. Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via assigning to `flags['FLAGNAME']` or *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.

`shape`
Tuple of array dimensions.

Examples

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

`strides`
Tuple of bytes to step in each dimension.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

Examples

```
>>> 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
```

`ndim`
Number of array dimensions.

Examples

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

`data`
Buffer object pointing to the start of the data.

`size`
Number of elements in the array.

Examples

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

`itemsize`
Length of one element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

`nbytes`
Number of bytes in the array.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

base

Base object if memory is from some other object.

Examples

Base of an array owning its memory is None:

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

Slicing creates a view, and the memory is shared with x:

```
>>> 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:

`ndarray.dtype`	Data-type for the array.

dtype

Data-type for the array.

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

Other attributes

`ndarray.T`	Same as self.transpose() except self is returned for self.ndim < 2.
`ndarray.real`	The real part of the array.
`ndarray.imag`	The imaginary part of the array.
`ndarray.flat`	A 1-d flat iterator.
`ndarray.ctypes`	A ctypes interface object.
`__array_priority__`	

T

Same as self.transpose() except self is returned for self.ndim < 2.

Examples

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

real
The real part of the array.

Examples

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

imag
The imaginary part of the array.

Examples

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

flat
A 1-d flat iterator.

Examples

```
>>> x = np.arange(3*4*5)
>>> x.shape = (3,4,5)
>>> x.flat[19]
19
>>> x.T.flat[19]
31
```

ctypes
A ctypes interface object.

Array interface

See Also:

The Array Interface.

`__array_interface__`	Python-side of the array interface
`__array_struct__`	C-side of the array interface

`ctypes` foreign function interface

`ndarray.ctypes`	A ctypes interface object.

ctypes
A ctypes interface object.

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 explained below.

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

`ndarray.item()`	Copy the first element of array to a standard Python scalar and return it. The array must be of size one.
`ndarray.tolist()`	Return the array as a possibly nested list.
`ndarray.itemset()`	
`ndarray.tostring([order])`	Construct a Python string containing the raw data bytes in the array.
`ndarray.tofile(fid[, sep, format])`	Write array to a file as text or binary.
`ndarray.dump(file)`	Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.
`ndarray.dumps()`	Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.
`ndarray.astype(t)`	Copy of the array, cast to a specified type.
`ndarray.byteswap(inplace)`	Swap the bytes of the array elements
`ndarray.copy([order])`	Return a copy of the array.
`ndarray.view([dtype, type])`	New view of array with the same data.
`ndarray.getfield(dtype, offset)`	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.
`ndarray.setflags([write,align,uic])`	
`ndarray.fill(value)`	Fill the array with a scalar value.

item()

Copy the first element of array to a standard Python scalar and return it. The array must be of size one.

tolist()

Return the array as a possibly nested list.

Return a copy of the array data as a hierarchical Python list. Data items are converted to the nearest compatible Python type.

Parameters

none :

Returns

y : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

itemset()

tostring(*order='C'*)

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

Parameters

order : {'C', 'F', None}

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

tofile(*fid, sep="", format="%s"*)

Write array to a file as text or binary.

Data is always written in 'C' order, independently of the order of *a*. The data produced by this method can be recovered by using the function fromfile().

This is a convenience function for quick storage of array data. Information on endianess 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 endianess. Some of these problems can be overcome by outputting the data as text files at the expense of speed and file size.

Parameters

fid : file or string

An open file object or a string containing a filename.

sep : string

Separator between array items for text output. If "" (empty), a binary file is written, equivalently to file.write(a.tostring()).

format : string

Format string for text file output. Each entry in the array is formatted to text by converting it to the closest Python type, and using "format" % item.

dump(*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file : str

A string naming the dump file.

dumps()

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

astype(*t*)

Copy of the array, cast to a specified type.

Parameters

t : string or dtype

Typecode or data-type to which the array is cast.

Examples

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

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

byteswap(*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace: bool, optional :

If `True`, swap bytes in-place, default is `False`.

Returns

out: ndarray :

The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

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

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
```

view(*dtype=None, type=None*)

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.

Examples

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

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> print y.dtype
int16
```

```
>>> print type(y)
<class 'numpy.core.defmatrix.matrix'>
```

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)
```

getfield(*dtype, offset*)

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.

setflags(*write=None, align=None, uic=None*)

fill(*value*)

Fill the array with a scalar value.

Parameters

a : ndarray

Input array

value : scalar

All elements of *a* will be assigned this value.

Examples

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

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.

`ndarray.reshape`(shape[, order])	Returns an array containing the same data with a new shape.
`ndarray.resize`(new_shape[, refcheck, order])	Change shape and size of array in-place.
`ndarray.transpose`(*axes)	Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.
`ndarray.swapaxes`(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
`ndarray.flatten`([order])	Collapse an array into one dimension.
`ndarray.ravel`([order])	Return a flattened array.
`ndarray.squeeze`()	Remove single-dimensional entries from the shape of *a*.

reshape(*shape, order='C'*)

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

Refer to `numpy.reshape` for full documentation.

See Also:

numpy.reshape
equivalent function

resize(*new_shape, refcheck=True, order=False*)
Change shape and size of array in-place.

Parameters

a : ndarray
Input array.

new_shape : {tuple, int}
Shape of resized array.

refcheck : bool, optional
If False, memory referencing will not be checked. Default is True.

order : bool, optional
<needs an explanation>. Default if False.

Returns

None :

Raises

ValueError :
If *a* does not own its own data, or references or views to it exist.

Examples

Shrinking an array: array is flattened in C-order, resized, and reshaped:

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

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

```
>>> b = np.array([[0,1],[2,3]])
>>> b.resize((2,3))
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing:

```
>>> c = a
>>> a.resize((1,1))
...
ValueError: cannot resize an array that has been referenced ...
```

transpose(**axes*)
Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

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

`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

`flatten` (*order='C'*)

Collapse an array 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.

Examples

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

`ravel` (*[order]*)

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See Also:

`numpy.ravel`
equivalent function

`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.

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

take (*indices, axis=None, out=None, mode='raise'*)
Return an array formed from the elements of a at the given indices.

Refer to `numpy.take` for full documentation.

See Also:

numpy.take
equivalent function

put (*indices, values, mode='raise'*)
Set a.flat[n] = values[n] for all n in indices.

Refer to `numpy.put` for full documentation.

See Also:

numpy.put
equivalent function

repeat (*repeats, axis=None*)
Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also:

numpy.repeat
equivalent function

choose (*choices, out=None, mode='raise'*)
Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also:

numpy.choose
equivalent function

sort (*axis=-1, kind='quicksort', order=None*)
Sort an array, in-place.

Parameters

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

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

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

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

`argsort`(*axis=-1, kind='quicksort', order=None*)
Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also:

`numpy.argsort`
equivalent function

`searchsorted`(*v, side='left'*)
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

`compress`(*condition, axis=None, out=None*)
Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also:

`numpy.compress`
equivalent function

`diagonal`(*offset=0, axis1=0, axis2=1*)
Return specified diagonals.

Refer to `numpy.diagonal` for full documentation.

See Also:

`numpy.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.
- 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).

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.

`ndarray.argmax`([axis,out])	Return indices of the maximum values along the given axis of *a*.
`ndarray.min`([axis,out])	Return the minimum along a given axis.
`ndarray.argmin`([axis,out])	Return indices of the minimum values along the given axis of *a*.
`ndarray.ptp`([axis,out])	Peak to peak (maximum - minimum) value along a given axis.
`ndarray.clip`(a_min,a_max[,out])	Return an array whose values are limited to `[a_min, a_max]`.
`ndarray.conj`()	Return an array with all complex-valued elements conjugated.
`ndarray.round`([decimals,out])	Return an array rounded a to the given number of decimals.
`ndarray.trace`([offset,axis1,axis2,...])	Return the sum along diagonals of the array.
`ndarray.sum`([axis,dtype,out])	Return the sum of the array elements over the given axis.
`ndarray.cumsum`([axis,dtype,out])	Return the cumulative sum of the elements along the given axis.
`ndarray.mean`([axis,dtype,out])	Returns the average of the array elements along given axis.
`ndarray.var`([axis,dtype,out,...])	Returns the variance of the array elements, along given axis.
`ndarray.std`([axis,dtype,out,...])	Returns the standard deviation of the array elements along given axis.
`ndarray.prod`([axis,dtype,out])	Return the product of the array elements over the given axis
`ndarray.cumprod`([axis,dtype,out])	Return the cumulative product of the elements along the given axis.
`ndarray.all`([axis,out])	Returns True if all elements evaluate to True.
`ndarray.any`([axis,out])	Check if any of the elements of *a* are true.

`argmax`(*axis=None, out=None*)

Return indices of the maximum values along the given axis of *a*.

Parameters

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.

Returns

index_array : ndarray

An array of indices or single index value, or a reference to *out* if it was specified.

Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

min (*axis=None, out=None*)
Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also:

`numpy.amin`
equivalent function

argmin (*axis=None, out=None*)
Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.ndarray.argmax` for detailed documentation.

ptp (*axis=None, out=None*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also:

`numpy.ptp`
equivalent function

clip (*a_min, a_max, out=None*)
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 ()
Return an array with all complex-valued elements conjugated.

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

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

sum (*axis=None, dtype=None, out=None*)
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 (*axis=None, dtype=None, out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also:

numpy.cumsum
equivalent function

mean (*axis=None, dtype=None, out=None*)
Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also:

numpy.mean
equivalent function

var (*axis=None, dtype=None, out=None, ddof=0*)
Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also:

numpy.var
equivalent function

std (*axis=None, dtype=None, out=None, ddof=0*)
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 (*axis=None, dtype=None, out=None*)
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 (*axis=None, dtype=None, out=None*)
Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also:

numpy.cumprod
equivalent function

all (*axis=None, out=None*)
Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

See Also:

numpy.all
equivalent function

any (*axis=None, out=None*)
Check if any of the elements of *a* are 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:

`ndarray.__lt__()`	x.__lt__(y) <==> x<y
`ndarray.__le__()`	x.__le__(y) <==> x<=y
`ndarray.__gt__()`	x.__gt__(y) <==> x>y
`ndarray.__ge__()`	x.__ge__(y) <==> x>=y
`ndarray.__eq__()`	x.__eq__(y) <==> x==y
`ndarray.__ne__()`	x.__ne__(y) <==> x!=y

__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`):

`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 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:

`ndarray.__neg__()`	x.__neg__() <==> -x
`ndarray.__pos__()`	x.__pos__() <==> +x
`ndarray.__abs__()<])`	
`ndarray.__invert__()`	x.__invert__() <==> ~x

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

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

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

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

Arithmetic:

`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)<,y)	
`ndarray.__pow__`(y[,z])<,y[,z])	
`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

`__add__`()
x.__add__(y) <==> x+y

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

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

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

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

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

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

`__divmod__`(*y) <==> divmod(x, y*)

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

`__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` only takes 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:

`ndarray.__iadd__`()	x.__iadd__(y) <==> x+y
`ndarray.__isub__`()	x.__isub__(y) <==> x-y
`ndarray.__imul__`()	x.__imul__(y) <==> x*y
`ndarray.__idiv__`()	x.__idiv__(y) <==> x/y
`ndarray.__itruediv__`()	x.__itruediv__(y) <==> x/y
`ndarray.__ifloordiv__`()	x.__ifloordiv__(y) <==> x//y
`ndarray.__imod__`()	x.__imod__(y) <==> x%y
`ndarray.__ipow__`()	x.__ipow__(y) <==> x**y
`ndarray.__ilshift__`()	x.__ilshift__(y) <==> x<<y
`ndarray.__irshift__`()	x.__irshift__(y) <==> x>>y
`ndarray.__iand__`()	x.__iand__(y) <==> x&y
`ndarray.__ior__`()	x.__ior__(y) <==> x|y
`ndarray.__ixor__`()	x.__ixor__(y) <==> x^y

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

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

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

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

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

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

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

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

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

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

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

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

__ixor__()
x.__ixor__(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:

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

Parameters
order : {'C', 'F', 'A'}, optional
If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

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

Used if copy.deepcopy is called on an array.

__reduce__()
For pickling.

__setstate__(*version, shape, dtype, isfortran, rawdata*)
For unpickling.

Parameters

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

shape : tuple

dtype : data-type

isFortran : bool

rawdata : string or list
a binary string with the data (or a list if 'a' is an object array)

Basic customization:

`ndarray.__new__()`	T.__new__(S, ...) -> a new object with type S, a subtype of T
`ndarray.__array__()`	a.__array__(|dtype) -> reference if type unchanged, copy otherwise.
`ndarray.__array_wrap__()`	a.__array_wrap__(obj) -> Object of same type as a from ndarray obj.

__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 a from ndarray obj.

Container customization: (see *Indexing*)

`ndarray.__len__()<])`	
`ndarray.__getitem__()`	x.__getitem__(y) <==> x[y]
`ndarray.__setitem__()`	x.__setitem__(i, y) <==> x[i]=y
`ndarray.__getslice__()`	x.__getslice__(i, j) <==> x[i:j]
`ndarray.__setslice__()`	x.__setslice__(i, j, y) <==> x[i:j]=y
`ndarray.__contains__()`	x.__contains__(y) <==> y in x

__len__() <==> *len(x)*

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

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.

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

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

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

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

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

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

String representations:

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

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

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

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`. [1] 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`).

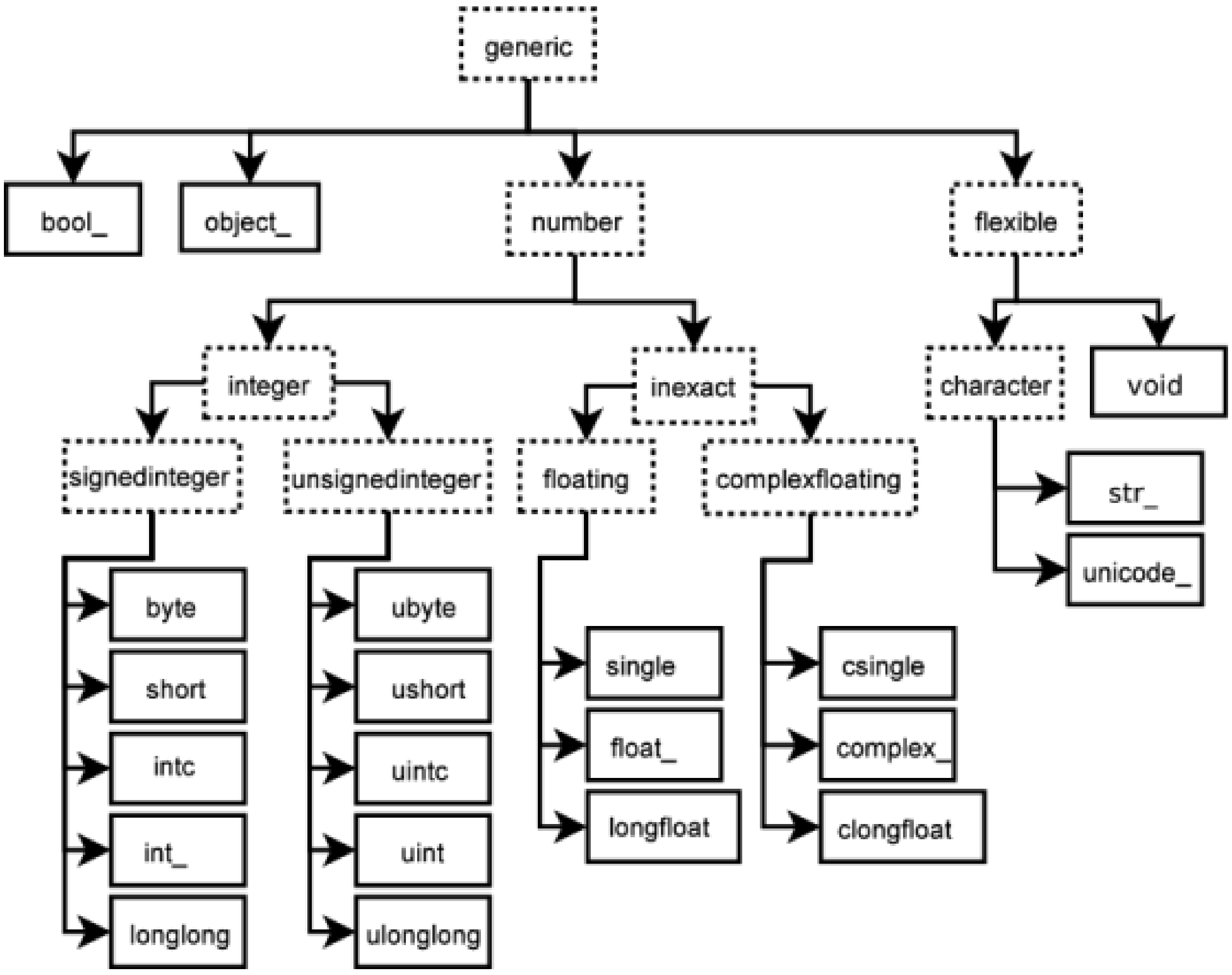

Figure 1.2: **Figure:** Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.

[1] However, array scalars are immutable, so that none of the array scalar attributes are settable.

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:

Array scalar type	Related Python type
`int_`	`IntType`
`float_`	`FloatType`
`complex_`	`ComplexType`
`str_`	`StringType`
`unicode_`	`UnicodeType`

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 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:

Type	Remarks	Character code
`bool_`	compatible: Python bool	`'?'`
`bool8`	8 bits	

Integers:

`byte`	compatible: C char	`'b'`
`short`	compatible: C short	`'h'`
`intc`	compatible: C int	`'i'`
`int_`	compatible: Python int	`'l'`
`longlong`	compatible: C long long	`'q'`
`intp`	large enough to fit a pointer	`'p'`
`int8`	8 bits	
`int16`	16 bits	
`int32`	32 bits	
`int64`	64 bits	

Unsigned integers:

`ubyte`	compatible: C unsigned char	`'B'`
`ushort`	compatible: C unsigned short	`'H'`
`uintc`	compatible: C unsigned int	`'I'`
`uint`	compatible: Python int	`'L'`
`ulonglong`	compatible: C long long	`'Q'`
`uintp`	large enough to fit a pointer	`'P'`
`uint8`	8 bits	
`uint16`	16 bits	
`uint32`	32 bits	
`uint64`	64 bits	

Floating-point numbers:

`single`	compatible: C float	`'f'`
`double`	compatible: C double	
`float_`	compatible: Python float	`'d'`
`longfloat`	compatible: C long float	`'g'`
`float32`	32 bits	
`float64`	64 bits	
`float96`	92 bits, platform?	
`float128`	128 bits, platform?	

Complex floating-point numbers:

`csingle`		`'F'`
`complex_`	compatible: Python complex	`'D'`
`clongfloat`		`'G'`
`complex64`	two 32-bit floats	
`complex128`	two 64-bit floats	
`complex192`	two 96-bit floats, platform?	
`complex256`	two 128-bit floats, platform?	

Any Python object:

`object_`	any Python object	`'O'`

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.)

`str_`	compatible: Python str	`'S#'`
`unicode_`	compatible: Python unicode	`'U#'`
`void`		`'V#'`

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`, `1 -> b`, `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.

Note: XXX: what to put in the type docstrings, and where to put them?

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:

`generic.flags`	integer value of flags
`generic.shape`	tuple of array dimensions
`generic.strides`	tuple of bytes steps in each dimension
`generic.ndim`	number of array dimensions
`generic.data`	pointer to start of data
`generic.size`	number of elements in the gentype
`generic.itemsize`	length of one element in bytes
`generic.base`	base object
`generic.dtype`	get array data-descriptor
`generic.real`	real part of scalar
`generic.imag`	imaginary part of scalar
`generic.flat`	a 1-d view of scalar
`generic.T`	transpose
`generic.__array_interface__`	Array protocol: Python side
`generic.__array_struct__`	Array protocol: struct
`generic.__array_priority__`	Array priority.
`generic.__array_wrap__()`	sc.__array_wrap__(obj) return scalar from array

`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

Note: XXX: import the documentation into the docstrings?

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['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:

generic	
generic.__array__()	sc.__array__(|type) return 0-dim array
generic.__array_wrap__()	sc.__array_wrap__(obj) return scalar from array
generic.__squeeze__	
generic.byteswap()	
generic.__reduce__()	
generic.__setstate__()	
generic.setflags()	

class **generic**()

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

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

byteswap()

__reduce__()

__setstate__()

setflags()

Note: XXX: import the documentation into the docstrings?

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:

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

Record, one field name 'f1', containing int16:

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

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

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

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

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

Using array-protocol type strings:

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

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

```
>>> np.dtype("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:

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

Subdivide `int16` into 2 `int8`'s, called x and y. 0 and 1 are the offsets in bytes:

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

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

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

Offsets in bytes, here 0 and 25:

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

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:

`number`, `inexact`, `floating`	`float`
`complexfloating`	`cfloat`
`integer`, `signedinteger`	`int_`
`unsignedinteger`	`uint`
`character`	`string`
`generic`, `flexible`	`void`

Built-in Python types

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

`int`	`int_`
`bool`	`bool_`
`float`	`float_`
`complex`	`cfloat`
`str`	`string`
`unicode`	`unicode_`
`buffer`	`void`
(all others)	`object_`

Example

```
>>> 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

```
>>> 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

`'b'`	Boolean
`'i'`	(signed) integer
`'u'`	unsigned integer
`'f'`	floating-point
`'c'`	complex-floating point
`'S'`, `'a'`	string
`'U'`	unicode
`'V'`	anything (`void`)

Example

```
>>> 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

```
>>> 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

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

Type strings

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

Example

```
>>> 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

```
>>> 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

```
>>> 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`.

```
>>> 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:

```
>>> 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:

```
>>> 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):

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

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

```
>>> 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:

```
>>> 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:

```
>>> 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):

```
>>> 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:

`dtype.type`	
`dtype.kind`	
`dtype.char`	
`dtype.num`	
`dtype.str`	

`type`

`kind`

`char`

`num`

`str`

Size of the data is in turn described by:

`dtype.name`	
`dtype.itemsize`	

`name`

`itemsize`

Endianness of this data:

`dtype.byteorder`	byteorder

`byteorder`

byteorder

String giving byteorder of dtype

One of: * '=' - native byteorder * '<' - little endian * '>' - big endian * '|' - endian not relevant

Examples

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

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

dtype.fields	
dtype.names	

fields

names

For data types that describe sub-arrays:

dtype.subdtype	
dtype.shape	

subdtype

shape

Attributes providing additional information:

dtype.hasobject	
dtype.flags	
dtype.isbuiltin	isbuiltin
dtype.isnative	
dtype.descr	
dtype.alignment	

hasobject

flags

isbuiltin
isbuiltin

Value identifying if numpy dtype is a numpy builtin type

Read-only

Returns

val : {0,1,2}

0 if this is a structured array type, with fields 1 if this is a dtype compiled into numpy (such as ints, floats etc) 2 if the dtype is for a user-defined numpy type

A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See the Guide to Numpy for details.

Examples

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

`isnative`

`descr`

`alignment`

Methods

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

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

`newbyteorder` (*new_order='S'*)

Return a new dtype with a different byte order.

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

Parameters

new_order : string, optional

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

- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- 'S' - swap dtype from current to opposite endian
- {'|', '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.

Examples

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

The following methods implement the pickle protocol:

`dtype.__reduce__()`	
`dtype.__setstate__()`	

`__reduce__``()`

`__setstate__``()`

1.4 Indexing

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

Note: In Python, `x[(exp1, exp2, ..., expN)]` is equivalent to `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 *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 *N* integers returns an *array scalar* representing the corresponding item. As in Python, all indices are zero-based: for the *i*-th index n_i, the valid range is $0 \le n_i < d_i$ 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 `i:j:k` where *i* is the starting index, *j* is the stopping index, and *k* is the step ($k \neq 0$). This selects the *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*.

  Example

  ```
  >>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  >>> x[1:7:2]
  array([1, 3, 5])
  ```

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

  Example

  ```
  >>> x[-2:10]
  array([8, 9])
  >>> x[-3:3:-1]
  array([7, 6, 5, 4])
  ```

- Assume *n* is the number of elements in the dimension being sliced. Then, if *i* is not given it defaults to 0 for *k > 0* and *n* for *k < 0* . If *j* is not given it defaults to *n* for *k > 0* and -1 for *k < 0* . If *k* is not given it defaults to 1. Note that `::` is the same as `:` and means select all indices along this axis.

  Example

  ```
  >>> 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

  ```
  >>> 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

```
>>> 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

  ```
  >>> 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 - 1*. If *N = 1* 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*, ..., *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] = 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 `[start:stop:step]` notation. For example, `x[1:10:5,::-1]` can also be implemented as `obj = (slice(1,10,5), slice(None,None,-1)); x[obj]` . This can be useful for constructing generic code that works on arrays of arbitrary dimension.

`newaxis`
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_t be the number of indexing arrays, and let $N_s = N - N_t$ be the number of slice objects. Note that $N_t > 0$ (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, ..., i_M)` over the range of the result shape and for each value of the index tuple `(ind_1, ..., ind_M)`:

  ```
  result[i_1, ..., i_M] == x[ind_1[i_1, ..., i_M], ind_2[i_1, ..., i_M],
                             ..., ind_N[i_1, ..., i_M]]
  ```

  Example

  Suppose the shape of the broadcasted indexing arrays is 3-dimensional and *N* is 2. Then the result is found by letting *i, j, k* run over the shape found by broadcasting `ind_1` and `ind_2`, and each *i, j, k* yields:

  ```
  result[i,j,k] = x[ind_1[i,j,k], ind_2[i,j,k]]
  ```

- 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 `ind` is a (2,3,4)-shaped indexing `intp` array, then `result = x[...,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 `result[...,i,j,k,:] = x[...,ind[i,j,k],:]`. This example produces the same result as `x.take(ind, axis=-2)`.

  Example

  Now let `x.shape` be (10,20,30,40,50) and suppose `ind_1` and `ind_2` are broadcastable to the shape (2,3,4). Then `x[:,ind_1,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[:,ind_1,:,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 `.transpose()` to move the subspace anywhere desired. (Note that this example cannot be replicated using `take`.)

Boolean

This advanced indexing occurs when 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[obj.nonzero()]` where, as described above, `obj.nonzero()` returns a tuple (of length `obj.ndim`) of integer index arrays showing the `True` elements of *obj*.

The special case when `obj.ndim == x.ndim` is worth mentioning. In this case `x[obj]` returns a 1-dimensional array filled with the elements of *x* corresponding to the `True` values of *obj*. The search order will be C-style (last index varies the fastest). If *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 `nonzero(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,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 `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['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['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

```
>>> 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

`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 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 arrayobject will not redefine certain aspects of the array object such as the buffer interface, or the attributes of the array. One of 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:

`__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_wrap__`(*array*)
This method should return an instance of the subclass from the `ndarray` object passed in. For example, this is called after every *ufunc* for the object with the highest array priority. The ufunc-computed array object is passed in and whatever is returned is passed to the user. Subclasses inherit a default implementation of this method.

`__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

`matrix.T`	transpose
`matrix.H`	hermitian (conjugate) transpose
`matrix.I`	inverse
`matrix.A`	base array

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.

`matrix`(data[,dtype,copy])	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).
`asmatrix`(data[,dtype])	Interpret the input as a matrix.
`bmat`(obj[,ldict,gdict])	Build a matrix object from a string, nested sequence, or array.

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]])
```

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]])
```

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

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

All the following expressions construct the same block matrix:

```
>>> np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
>>> np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
>>> np.bmat('A,B; C,D')
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```

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.

`memmap`	Create a memory-map to an array stored in a file on disk.
`memmap.flush`(self)	Flush any changes in the array to the file on disk.

class **memmap**()

Create a memory-map to an array stored in a 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 : string or file-like object

The file name or file object to be used as the array data buffer.

dtype : data-type, optional

The data-type used to interpret the file contents. Default is *uint8*

mode : {'r+', 'r', 'w+', 'c'}, optional

The file is opened in this mode:

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

Default is 'r+'.

offset : integer, optional

In the file, array data starts at this offset. *offset* 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 defaullt order is 'C'.

Methods

close :

Close the memmap file.

flush :

Flush any changes in memory to file on disk. When you delete a memmap object, flush is called first to write changes to disk before removing the object.

Notes

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.

Examples

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

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

```
>>> from tempfile import mkdtemp
>>> import os.path as path
>>> filename = path.join(mkdtemp(), 'newfile.dat')
```

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

```
>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
>>> fp
memmap([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]], dtype=float32)
```

Write data to memmap array:

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

Deletion flushes memory changes to disk before removing the object:

```
>>> del fp
```

Load the memmap and verify data was stored:

```
>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> newfp
memmap([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
```

Read-only memmap:

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

Cannot assign to read-only, obviously:

```
>>> fpr[0, 3] = 56
Traceback (most recent call last):
    ...
RuntimeError: array is not writeable
```

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)
```

flush()
Flush any changes in the array to the file on disk.

Example:

```
>>> 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)

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 string 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,...])	A character array of string or unicode type.
`core.defchararray.array`(obj[,itemsize,copy,...])	

class **`chararray`**()

A character array of string or unicode type.

The array items will be *itemsize* characters long.

Create the array using buffer (with offset and strides) if it is not None. If buffer is None, then construct a new array with strides in Fortran order if len(shape) >=2 and order is 'Fortran' (otherwise the strides will be in 'C' order).

`array`(*obj, itemsize=None, copy=True, unicode=False, order=None*)

Another difference with the standard ndarray of string 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`.

`recarray`	Construct an ndarray that allows field access using attributes.
`record`	A data-type scalar that allows field access as attribute lookup.

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']`. *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 strings, 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 reccord array from data, use one of the following methods:

1.Create a standard ndarray and convert it to a record array, using `arr.view(np.recarray)`

2.Use the *buf* keyword.

3.Use *np.rec.fromrecords*.

Examples

Create an array with two fields, `x` and `y`:

```
>>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', float), ('y', int)])
>>> x
array([(1.0, 2), (3.0, 4)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

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

View the array as a record array:

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

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

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

Create a new, empty record array:

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

class **`record`**()
A data-type scalar that allows field access as attribute lookup.

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.

`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 *myiter* is an iterator object, then the Python code:

```
for val in myiter:
    ...
    some code involving val
    ...
```

calls `val = myiter.next()` repeatedly until `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:

```
for i in arr.shape[0]:
    val = arr[i]
```

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.

```
>>> 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]]
```

Flat iteration

`ndarray.flat`	A 1-d flat iterator.

`flat`

A 1-d flat iterator.

Examples

```
>>> x = np.arange(3*4*5)
>>> x.shape = (3,4,5)
>>> x.flat[19]
19
>>> x.T.flat[19]
31
```

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

```
>>> 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

`ndenumerate`	Multidimensional index iterator.

class `ndenumerate`(*arr*)

Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

Parameters

a : ndarray

Input array.

Examples

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

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

```
>>> for i, val in ndenumerate(a):
...     if sum(i)%5 == 0: print i, val
(0, 0, 0) 10
(1, 1, 3) 25
(2, 0, 3) 29
(2, 1, 2) 32
```

Iterator for broadcasting

broadcast	

class broadcast()

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.

```
>>> 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:

```
>>> import numpy as np
>>> 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:

```
>>> 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:

```
>>> 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`:

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

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

```
>>> 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:

```
>>> 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`.

`array`(data[,dtype,copy,order,...])	Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.
`masked_array`	Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

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*)

Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction:
x = MaskedArray(data, mask=nomask, dtype=None, copy=True, fill_value=None, keep_mask=True, hard_mask=False, shrink=True)

Parameters

data : {var}
Input data.

mask : {nomask, sequence}, optional
Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., 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 : {False, True}, optional
Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, False}, optional
Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}, optional
Minimum number of dimensions

fill_value : {var}, optional
Value used to fill in the masked values when necessary. If None, a default based on the datatype is used.

keep_mask : {True, boolean}, optional
Whether to combine mask with the mask of the input data, if any (True), or to use only mask for the output (False).

hard_mask : {False, boolean}, optional
Whether to use a hard mask or not. With a hard mask, masked values cannot be unmasked.

shrink : {True, boolean}, optional
Whether to force compression of an empty mask.

class `masked_array()`
Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction:
x = MaskedArray(data, mask=nomask, dtype=None, copy=True, fill_value=None, keep_mask=True, hard_mask=False, shrink=True)

Parameters

data : {var}
Input data.

mask : {nomask, sequence}, optional
Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., 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 : {False, True}, optional
Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, False}, optional

Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}, optional

Minimum number of dimensions

fill_value : {var}, optional

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

keep_mask : {True, boolean}, optional

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

hard_mask : {False, boolean}, optional

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

shrink : {True, boolean}, optional

Whether to force compression of an empty mask.

- 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.

```
>>> 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:

`asarray`(a[,dtype,order])	Convert the input *a* to a masked array of the given datatype.
`asanyarray`(a[,dtype])	Convert the input *a* to a masked array of the given datatype. If *a* is a subclass of MaskedArray, its class is conserved.
`fix_invalid`(a[,mask,copy,fill_value])	Return (a copy of) *a* where invalid data (nan/inf) are masked and replaced by *fill_value*.
`masked_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x == value)`.
`masked_greater`(x,value[,copy])	Return the array *x* masked where `(x > value)`. Any value of mask already masked is kept masked.
`masked_greater_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x >= value)`.
`masked_inside`(x,v1,v2[,copy])	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.
`masked_invalid`(a[,copy])	Mask the array for invalid values (NaNs or infs). Any preexisting mask is conserved.
`masked_less`(x,value[,copy])	Shortcut to masked_where, with condition `(x < value)`.
`masked_less_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x <= value)`.
`masked_not_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x != value)`.
`masked_object`(x,value[,copy,shrink])	Mask the array *x* where the data are exactly equal to value.
`masked_outside`(x,v1,v2[,copy])	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.
`masked_values`(x,value[,rtol,atol,copy,...])	Mask the array *x* where the data are approximately equal in value, i.e. `(abs(x - value) <= atol+rtol*abs(value))`
`masked_where`(condition,a[,copy])	Return *a* as an array masked where `condition` is `True`. Masked values of `a` or `condition` are kept.

asarray(*a, dtype=None, order=None*)

Convert the input *a* to a masked array of the given datatype.

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 ('FORTRAN') memory representation. Defaults to 'C'.

Returns

out : ndarray

MaskedArray interpretation of *a*. No copy is performed if the input is already an ndarray. If *a* is a subclass of MaskedArray, a base class MaskedArray is returned.

asanyarray(*a, dtype=None*)

Convert the input *a* to a masked array of the given datatype. If *a* is a subclass of MaskedArray, its class is conserved.

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 ('FORTRAN') memory representation. Defaults to 'C'.

Returns

out : ndarray

MaskedArray interpretation of *a*. No copy is performed if the input is already an ndarray.

fix_invalid(*a, mask=False, copy=True, fill_value=None*)

Return (a copy of) *a* where invalid data (nan/inf) are masked and replaced by *fill_value*.

Note that a copy is performed by default (just in case...).

Parameters

a : array_like

A (subclass of) ndarray.

copy : bool

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

fill_value : {var}, optional

Value used for fixing invalid data. If not given, the output of get_fill_value(a) is used instead.

Returns

b : MaskedArray

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

Shortcut to masked_where, with condition `(x == value)`.

See Also:

masked_where
base function

masked_values
equivalent function for floats.

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

Return the array *x* masked where `(x > value)`. Any value of mask already masked is kept masked.

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

Shortcut to masked_where, with condition `(x >= value)`.

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

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.

Notes

The array x is prefilled with its filling value.

masked_invalid(*a, copy=True*)

Mask the array for invalid values (NaNs or infs). Any preexisting mask is conserved.

masked_less (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x < value)`.

masked_less_equal (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x <= value)`.

masked_not_equal (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x != value)`.

masked_object (*x, value, copy=True, shrink=True*)
Mask the array *x* where the data are exactly equal to value.

This function is suitable only for object arrays: for floating point, please use **'masked_values'_** instead.

Parameters

x : array_like
Array to mask

value : var
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

masked_outside (*x, v1, v2, copy=True*)
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.

Notes

The array x is prefilled with its filling value.

masked_values (*x, value, rtol=1.0000000000000001e-05, atol=1e-08, copy=True, shrink=True*)
Mask the array x where the data are approximately equal in value, i.e. `(abs(x - value) <= atol+rtol*abs(value))`

Suitable only for floating points. For integers, please use `masked_equal`. The mask is set to `nomask` if posible.

Parameters

x : array_like
Array to fill.

value : float
Masking value.

rtol : {float}, optional
Tolerance parameter.

atol : {float}, optional
Tolerance parameter (1e-8).

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

masked_where (*condition, a, copy=True*)
Return `a` as an array masked where `condition` is `True`. Masked values of `a` or `condition` are kept.

Parameters

condition : array_like
Masking condition.

a : array_like

Array to mask.

copy : bool

Whether to return a copy of a (True) or modify a in place (False).

Accessing the data

The underlying data of a masked array can be accessed through 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, invalid data should not be relied on. If a representation of the array is needed 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:

```
>>> 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):

```
>>> 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:

```
>>> x = ma.array([1, 2, 3])
>>> x[0] = ma.masked
>>> x
masked_array(data = [-- 2 3],
             mask = [ True False False],
       fill_value = 999999)
>>> 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)
>>> z = ma.array([1, 2, 3, 4])
>>> 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 not work if the masked array has a *hard* mask, as shown by the `hardmask` attribute. This feature was introduced to prevent the overwriting of the mask. To force the unmasking of an entry in such circumstance, the mask has first to be softened with the `soften_mask` method before the allocation, and then re-hardened with `harden_mask`:

```
>>> 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 --],
             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.soften_mask()
```

To unmask all masked entries of a masked array, the simplest solution is to assign the constant `nomask` to the mask:

```
>>> 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 = 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`):

```
>>> 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.

```
>>> 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.

```
>>> 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 invalid data may actually be affected by the operation in some cases and once again that invalid data should not be relied on.

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:

```
>>> 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:

```
>>> 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):

```
>>> 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.

```
>>> 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.:

```
>>> 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]`:

```
>>> 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:

```
>>> 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 array 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.

```
>>> 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.

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:

```
>>> x = 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)])
>>> x.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 0d-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:

datatype	default
bool	True
int	999999
float	1.e20
complex	1.e20+0j
object	'?'
string	'N/A'

baseclass

Returns the class of the underlying data.

```
>>> x =  ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 0], [1, 0]])
>>> x.baseclass
<class 'numpy.core.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.

`MaskedArray.base`	Base object if memory is from some other object.
`MaskedArray.ctypes`	A ctypes interface object.
`MaskedArray.dtype`	Data-type for the array.
`MaskedArray.flags`	Information about the memory layout of the array.
`MaskedArray.itemsize`	Length of one element in bytes.
`MaskedArray.nbytes`	Number of bytes in the array.
`MaskedArray.ndim`	Number of array dimensions.
`MaskedArray.shape`	Tuple of array dimensions.
`MaskedArray.size`	Number of elements in the array.
`MaskedArray.strides`	Tuple of bytes to step in each dimension.
`MaskedArray.imag`	Imaginary part.
`MaskedArray.real`	Real part
`MaskedArray.flat`	Flat version of the array.
`MaskedArray.__array_priority__`	int(x[, base]) -> integer

base
Base object if memory is from some other object.

Examples

Base of an array owning its memory is None:

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

Slicing creates a view, and the memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

ctypes
A ctypes interface object.

dtype
Data-type for the array.

flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C) :
The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F) :
The data is in a single, Fortran-style contiguous segment.

OWNDATA (O) :
The array owns the memory it uses or borrows it from another object.

WRITEABLE (W) :
The data area can be written to.

ALIGNED (A) :
The data and strides are aligned appropriately for the hardware.

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

FNC :
F_CONTIGUOUS and not C_CONTIGUOUS.

FORC :
F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B) :
ALIGNED and WRITEABLE.

CARRAY (CA) :
BEHAVED and C_CONTIGUOUS.

FARRAY (FA) :
BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be also accessed dictionary-like, and using lowercased attribute names. Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via assigning to `flags['FLAGNAME']` or *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.

itemsize

Length of one element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

nbytes

Number of bytes in the array.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

Number of array dimensions.

Examples

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

shape

Tuple of array dimensions.

Examples

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

size

Number of elements in the array.

Examples

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

strides

Tuple of bytes to step in each dimension.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

Examples

```
>>> 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

MaskedArray.__float__(self)	Convert to float.
MaskedArray.__hex__()<])	
MaskedArray.__int__(self)	Convert to int.
MaskedArray.__long__()<])	
MaskedArray.__oct__()<])	
MaskedArray.view([dtype,type])	New view of array with the same data.
MaskedArray.astype(self,newtype)	Returns a copy of the MaskedArray cast to given newtype.
MaskedArray.byteswap(inplace)	Swap the bytes of the array elements
MaskedArray.compressed(self)	Return a 1-D array of all the non-masked data.
MaskedArray.filled(self[,fill_value])	Return a copy of self, where masked values are filled with *fill_value*.
MaskedArray.tofile(self,fid[,sep,format])	
MaskedArray.toflex(self)	Transforms a MaskedArray into a flexible-type array with two fields:
MaskedArray.tolist(self[,fill_value])	Copy the data portion of the array to a hierarchical python list and returns that list.
MaskedArray.torecords(self)	Transforms a MaskedArray into a flexible-type array with two fields:
MaskedArray.tostring(self[,fill_value,order])	Return a copy of array data as a Python string containing the raw bytes in the array. The array is filled beforehand.

__float__()
Convert to float.

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

__int__()
Convert to int.

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

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

view(*dtype=None, type=None*)
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.

Examples

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

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> print y.dtype
int16

>>> print type(y)
<class 'numpy.core.defmatrix.matrix'>
```

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

```
>>> 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]
 [-- 5 --]
 [7 -- 9]]
```

byteswap(*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace: bool, optional :

If `True`, swap bytes in-place, default is `False`.

Returns

out: ndarray :

The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

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

compressed()

Return a 1-D array of all the non-masked data.

Returns

data : ndarray.

A new ndarray holding the non-masked data is returned.

Notes

- The result is NOT a MaskedArray !

Examples

```
>>> x = array(arange(5), mask=[0]+[1]*4)
>>> print x.compressed()
[0]
>>> print type(x.compressed())
<type 'numpy.ndarray'>
```

filled (*fill_value=None*)

Return a copy of self, where masked values are filled with *fill_value*.

If *fill_value* is None, *self.fill_value* is used instead.

Notes

- Subclassing is preserved
- The result is NOT a MaskedArray !

Examples

```
>>> 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'>
```

tofile (*fid, sep=", format='%s'*)

toflex ()

Transforms a MaskedArray into a flexible-type array with two fields:

- the `_data` field stores the `_data` part of the array;
- the `_mask` field stores the `_mask` part of the array;

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

```
>>> 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.torecords()
[[(1, False) (2, True) (3, False)]
 [(4, True) (5, False) (6, True)]
 [(7, False) (8, True) (9, False)]]
```

tolist (*fill_value=None*)

Copy the data portion of the array to a hierarchical python list and returns that 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`.

torecords ()

Transforms a MaskedArray into a flexible-type array with two fields:

- the `_data` field stores the `_data` part of the array;

•the `_mask` field stores the `_mask` part of the array;

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

```
>>> 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.torecords()
[[(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 a copy of array data as a Python string containing the raw bytes in the array. The array is filled beforehand.

Parameters

fill_value : {var}, optional

Value used to fill in the masked values. If None, uses self.fill_value instead.

order : {string}

Order of the data item in the copy {'C','F','A'}. 'C' – C order (row major) 'Fortran' – Fortran order (column major) 'Any' – Current order of array. None – Same as "Any"

Notes

As for method:*ndarray.tostring*, information about the shape, dtype..., but also fill_value will be lost.

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.

`MaskedArray.flatten`([order])	Collapse an array into one dimension.
`MaskedArray.ravel`(self)	Returns a 1D version of self, as a view.
`MaskedArray.reshape`(self, *s, **kwargs)	Returns a masked array containing the data of a, but with a new shape. The result is a view to the original array; if this is not possible, a ValueError is raised.
`MaskedArray.resize`(self, newshape[, refcheck, order])	Change shape and size of array in-place.
`MaskedArray.squeeze`()	Remove single-dimensional entries from the shape of *a*.
`MaskedArray.swapaxes`(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
`MaskedArray.transpose`(*axes)	Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.
`MaskedArray.T`	

`flatten`(*order='C'*)

Collapse an array 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.

Examples

```
>>> 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

```
>>> 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*)

Returns a masked array containing the data of a, but with a new shape. The result is a view to the original array; if this is not possible, a ValueError is raised.

Parameters

shape : shape tuple or int

The new shape should be compatible with the original shape. If an integer, then the result will be a 1D array of that length.

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 : array

A new view to the array.

Notes

If you want to modify the shape in place, please use `a.shape = s`

Examples

```
>>> 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*)

Change shape and size of array in-place.

squeeze()

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

Refer to `numpy.squeeze` for full documentation.

See Also:

numpy.squeeze
equivalent function

swapaxes(*axis1, axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also:

numpy.swapaxes
equivalent function

transpose(**axes*)

Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

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

T

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.

`MaskedArray.argmax`(self[,axis,fill_value,out])	Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.
`MaskedArray.argmin`(self[,axis,fill_value,out])	Return array of indices to the minimum values along the given axis.
`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.
`MaskedArray.choose`(choices[,out,mode])	Use an index array to construct a new array from a set of choices.
`MaskedArray.compress`(self,condition[,axis,out])	Return *a* where condition is `True`. If condition is a *MaskedArray*, missing values are considered as `False`.
`MaskedArray.diagonal`([offset,axis1,axis2])	Return specified diagonals.
`MaskedArray.fill`(value)	Fill the array with a scalar value.
`MaskedArray.item`()	Copy the first element of array to a standard Python scalar and return it. The array must be of size one.
`MaskedArray.nonzero`(self)	Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.
`MaskedArray.put`(self,indices,values[,mode])	Set storage-indexed locations to corresponding values.
`MaskedArray.repeat`(repeats[,axis])	Repeat elements of an array.
`MaskedArray.searchsorted`(v[,side])	Find indices where elements of v should be inserted in a to maintain order.
`MaskedArray.sort`(self[,axis,kind,order,...])	Return a sorted copy of an array.
`MaskedArray.take`(indices[,axis,out,mode])	Return an array formed from the elements of a at the given indices.

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. If not given, 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.

`choose`(*choices, out=None, mode='raise'*)
Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also:

`numpy.choose`
equivalent function

`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.

Warning: Please note the difference with `compressed` ! The output of `compress` has a mask, the output of `compressed` does not.

`diagonal`(*offset=0, axis1=0, axis2=1*)
Return specified diagonals.

Refer to `numpy.diagonal` for full documentation.

See Also:

`numpy.diagonal`
equivalent function

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

Parameters
a : ndarray
Input array
value : scalar
All elements of *a* will be assigned this value.

Examples

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

item()

Copy the first element of array to a standard Python scalar and return it. The array must be of size one.

nonzero()

Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.

There are as many tuples as dimensions of a, each tuple contains 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: `transpose(a.nonzero())`.

The result of this is always a 2d array, with a row for each non-zero element.

put(*indices, values, mode='raise'*)

Set 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

```
>>> 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 (*repeats, axis=None*)
Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also:

numpy.repeat
equivalent function

searchsorted (*v, side='left'*)
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 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:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```
>>> 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:

```
>>> 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')])
```

take(*indices, axis=None, out=None, mode='raise'*)
Return an array formed from the elements of a at the given indices.

Refer to `numpy.take` for full documentation.

See Also:

numpy.take
equivalent function

Pickling and copy

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 (*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
```

dump (*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file : str

A string naming the dump file.

dumps ()

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

Calculations

`MaskedArray.all`(self[, axis, out])	Check if all of the elements of *a* are true.
`MaskedArray.anom`(self[, axis, dtype])	Return the anomalies (deviations from the average) along the given axis.
`MaskedArray.any`(self[, axis, out])	Check if any of the elements of *a* are true.
`MaskedArray.clip`(a_min, a_max[, out])	Return an array whose values are limited to `[a_min, a_max]`.
`MaskedArray.conj`()	Return an array with all complex-valued elements conjugated.
`MaskedArray.conjugate`()	Return an array with all complex-valued elements conjugated.
`MaskedArray.cumprod`(self[, axis, dtype, out])	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.
`MaskedArray.cumsum`(self[, axis, dtype, out])	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.
`MaskedArray.max`(self[, axis, out, fill_value])	Return the maximum along a given axis.
`MaskedArray.mean`(self[, axis, dtype, out])	Returns the average of the array elements along given axis. Refer to `numpy.mean` for full documentation.
`MaskedArray.min`(self[, axis, out, fill_value])	Return the minimum along a given axis.
`MaskedArray.prod`(self[, axis, dtype, out])	Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.
`MaskedArray.product`(self[, axis, dtype, out])	Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.
`MaskedArray.ptp`(self[, axis, out, fill_value])	Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).
`MaskedArray.round`([decimals, out])	Return an array rounded a to the given number of decimals.
`MaskedArray.std`(self[, axis, dtype, out])	Compute the standard deviation along the specified axis.
`MaskedArray.sum`(self[, axis, dtype, out])	Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
`MaskedArray.trace`([offset, axis1, axis2, ...])	Return the sum along diagonals of the array.
`MaskedArray.var`(self[, axis, dtype, out])	Compute the variance along the specified axis.

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

```
>>> 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*)

Return the anomalies (deviations from the average) along the given axis.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

dtype : {dtype}, optional

Datatype for the intermediary computation. If not given, the current dtype is used instead.

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 (*a_min, a_max, out=None*)

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()
Return an array with all complex-valued elements conjugated.

conjugate()
Return an array with all complex-valued elements conjugated.

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

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.

Warning: The mask is lost if out is not a valid `MaskedArray` !

Notes

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]
```

`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 along given axis. Refer to `numpy.mean` for full documentation.

See Also:

`numpy.mean`

equivalent function'

`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.])
```

`product`(*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.])
```

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 - ddof`, where `N` represents the number of elements. By default *ddof* is zero (biased estimate).

Returns

standard_deviation : {ndarray, scalar}; 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:

`numpy.var`
Variance

`numpy.mean`
Average

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `var = sqrt(mean(abs(x - x.mean())**2))`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, 0)
array([ 1.,  1.])
>>> np.std(a, 1)
array([ 0.5,  0.5])
```

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

```
>>> 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 - 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

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of 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.

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])
```

Arithmetic and comparison operations

Comparison operators:

`MaskedArray.__lt__()`	x.__lt__(y) <==> x<y
`MaskedArray.__le__()`	x.__le__(y) <==> x<=y
`MaskedArray.__gt__()`	x.__gt__(y) <==> x>y
`MaskedArray.__ge__()`	x.__ge__(y) <==> x>=y
`MaskedArray.__eq__(self,other)`	Check whether other equals self elementwise
`MaskedArray.__ne__(self,other)`	Check whether other doesn't equal self elementwise

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

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

Arithmetic:

MaskedArray.__abs__()<])	
MaskedArray.__add__(self,other)	Add other to self, and return a new masked array.
MaskedArray.__radd__(self,other)	Add other to self, and return a new masked array.
MaskedArray.__sub__(self,other)	Subtract other to self, and return a new masked array.
MaskedArray.__rsub__(self,other)	Subtract other to self, and return a new masked array.
MaskedArray.__mul__(self,other)	Multiply other by self, and return a new masked array.
MaskedArray.__rmul__(self,other)	Multiply other by self, and return a new masked array.
MaskedArray.__div__(self,other)	Divide other into self, and return a new masked array.
MaskedArray.__rdiv__()	x.__rdiv__(y) <==> y/x
MaskedArray.__truediv__(self,other)	Divide other into self, and return a new masked array.
MaskedArray.__rtruediv__()	x.__rtruediv__(y) <==> y/x
MaskedArray.__floordiv__(self,other)	Divide other into self, and return a new masked array.
MaskedArray.__rfloordiv__()	x.__rfloordiv__(y) <==> y//x
MaskedArray.__mod__()	x.__mod__(y) <==> x%y
MaskedArray.__rmod__()	x.__rmod__(y) <==> y%x
MaskedArray.__divmod__(y)<,y)	
MaskedArray.__rdivmod__(y)<,x)	
MaskedArray.__pow__(self,other)	Raise self to the power other, masking the potential NaNs/Infs
MaskedArray.__rpow__(x[,z])<,y[,z])	
MaskedArray.__lshift__()	x.__lshift__(y) <==> x<<y
MaskedArray.__rlshift__()	x.__rlshift__(y) <==> y<<x
MaskedArray.__rshift__()	x.__rshift__(y) <==> x>>y
MaskedArray.__rrshift__()	x.__rrshift__(y) <==> y>>x
MaskedArray.__and__()	x.__and__(y) <==> x&y
MaskedArray.__rand__()	x.__rand__(y) <==> y&x
MaskedArray.__or__()	x.__or__(y) <==> xly
MaskedArray.__ror__()	x.__ror__(y) <==> ylx
MaskedArray.__xor__()	x.__xor__(y) <==> x^y
MaskedArray.__rxor__()	x.__rxor__(y) <==> y^x

`__abs__`() <==> *abs(x)*

`__add__`(*other*)
Add other to self, and return a new masked array.

`__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__`(*y) <==> divmod(x, y*)

`__rdivmod__`(*y) <==> divmod(y, x*)

`__pow__`(*other*)
Raise self to the power other, masking the potential NaNs/Infs

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

`__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:

`MaskedArray.__iadd__`(self,other)	Add other to self in-place.
`MaskedArray.__isub__`(self,other)	Subtract other from self in-place.
`MaskedArray.__imul__`(self,other)	Multiply self by other in-place.
`MaskedArray.__idiv__`(self,other)	Divide self by other in-place.
`MaskedArray.__itruediv__`()	x.__itruediv__(y) <==> x/y
`MaskedArray.__ifloordiv__`()	x.__ifloordiv__(y) <==> x//y
`MaskedArray.__imod__`()	x.__imod__(y) <==> x%y
`MaskedArray.__ipow__`(self,other)	Raise self to the power other, in place.
`MaskedArray.__ilshift__`()	x.__ilshift__(y) <==> x<<y
`MaskedArray.__irshift__`()	x.__irshift__(y) <==> x>>y
`MaskedArray.__iand__`()	x.__iand__(y) <==> x&y
`MaskedArray.__ior__`()	x.__ior__(y) <==> x|y
`MaskedArray.__ixor__`()	x.__ixor__(y) <==> x^y

__iadd__(*other*)
Add other to self in-place.

__isub__(*other*)
Subtract other from self in-place.

__imul__(*other*)
Multiply self by other in-place.

__idiv__(*other*)

Divide self by other in-place.

`__itruediv__``()`
x.__itruediv__(y) <==> x/y

`__ifloordiv__``()`
x.__ifloordiv__(y) <==> x//y

`__imod__``()`
x.__imod__(y) <==> x%y

`__ipow__``(`*other*`)`
Raise self to the power other, in place.

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

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

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

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

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

Representation

`MaskedArray.__repr__`(self)	Literal string representation.
`MaskedArray.__str__`(self)	String representation.
`MaskedArray.ids`(self)	Return the addresses of the data and mask areas.
`MaskedArray.iscontiguous`(self)	Is the data contiguous?

`__repr__``()`
Literal string representation.

`__str__``()`
String representation.

`ids``()`
Return the addresses of the data and mask areas.

`iscontiguous``()`
Is the data contiguous?

Special methods

For standard library functions:

MaskedArray.__copy__([order])	Return a copy of the array.
MaskedArray.__deepcopy__(self[,memo])	
MaskedArray.__getstate__(self)	Return the internal state of the masked array, for pickling purposes.
MaskedArray.__reduce__(self)	Return a 3-tuple for pickling a MaskedArray.
MaskedArray.__setstate__(self,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:

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

Parameters
order : {'C', 'F', 'A'}, optional
If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

__deepcopy__(*memo=None*)

__getstate__()
Return the internal state of the masked array, for pickling purposes.

__reduce__()
Return a 3-tuple for pickling a 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:

- 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.

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.
MaskedArray.__array__()	a.__array__(|dtype) -> reference if type unchanged, copy otherwise.
MaskedArray.__array_wrap__(self,obj[,context])	Special hook for ufuncs. Wraps the numpy array and sets the mask according to context.

__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__(*obj, context=None*)
Special hook for ufuncs. Wraps the numpy array and sets the mask according to context.

Container customization: (see *Indexing*)

`MaskedArray.__len__`()<])	
`MaskedArray.__getitem__`(self,indx)	x.__getitem__(y) <==> x[y]
`MaskedArray.__setitem__`(self,indx,value)	x.__setitem__(i, y) <==> x[i]=y
`MaskedArray.__delitem__`()	x.__delitem__(y) <==> del x[y]
`MaskedArray.__getslice__`(self,i,j)	x.__getslice__(i, j) <==> x[i:j]
`MaskedArray.__setslice__`(self,i,j,value)	x.__setslice__(i, j, value) <==> x[i:j]=value
`MaskedArray.__contains__`()	x.__contains__(y) <==> y in x

`__len__`() <==> *len(x)*

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

Return the item described by i, as a masked array.

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

Set item described by index. If value is masked, masks those locations.

`__delitem__`()
x.__delitem__(y) <==> del x[y]

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

Return the slice described by (i, j). The use of negative indices is not supported.

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

Set the slice (i,j) of a to value. If value is masked, mask those locations.

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

Specific methods

Handling the mask

The following methods can be used to access information about the mask or to manipulate the mask.

`MaskedArray.__setmask__`(self,mask[,copy])	Set the mask.
`MaskedArray.harden_mask`(self)	Force the mask to hard.
`MaskedArray.soften_mask`(self)	Force the mask to soft.
`MaskedArray.unshare_mask`(self)	Copy the mask and set the sharedmask flag to False.
`MaskedArray.shrink_mask`(self)	Reduce a mask to nomask when possible.

`__setmask__`(*mask, copy=False*)
Set the mask.

`harden_mask`()
Force the mask to hard.

`soften_mask`()
Force the mask to soft.

`unshare_mask`()
Copy the mask and set the sharedmask flag to False.

`shrink_mask`()
Reduce a mask to nomask when possible.

Handling the *fill_value*

`MaskedArray.get_fill_value`(self)	Return the filling value.
`MaskedArray.set_fill_value`(self[,value])	Set the filling value to value.

`get_fill_value`()
Return the filling value.

`set_fill_value`(*value=None*)
Set the filling value to value.

If value is None, use a default based on the data type.

Counting the missing elements

`MaskedArray.count`(self[,axis])	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 the non masked elements are counted.

Returns
result : MaskedArray
A masked array where the mask is True where all data are masked. If axis is None, returns either a scalar ot the masked singleton if all values are masked.

1.6.7 Masked array operations

Constants

`ma.MaskType`	

class **`MaskType`**()

Creation

From existing data

`ma.masked_array`	Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.
`ma.array`(data[,dtype,copy,order,...])	Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.
`ma.copy`()	copy a.copy(order='C')
`ma.frombuffer`(buffer[,dtype,count,offset])	Interpret a buffer as a 1-dimensional array.
`ma.fromfunction`(function,shape,**kwargs)	Construct an array by executing a function over each coordinate.
`ma.MaskedArray.copy`([order])	Return a copy of the array.

class **`masked_array`**()

Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction:

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

Parameters

data : {var}

Input data.

mask : {nomask, sequence}, optional

Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., 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 : {False, True}, optional

Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, False}, optional

Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}, optional

Minimum number of dimensions

fill_value : {var}, optional

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

keep_mask : {True, boolean}, optional

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

hard_mask : {False, boolean}, optional

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

shrink : {True, boolean}, optional

Whether to force compression of an empty mask.

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*)

Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction:

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

Parameters

data : {var}

Input data.

mask : {nomask, sequence}, optional

Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., 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 : {False, True}, optional

Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, False}, optional

Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}, optional

Minimum number of dimensions

fill_value : {var}, optional

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

keep_mask : {True, boolean}, optional

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

hard_mask : {False, boolean}, optional

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

shrink : {True, boolean}, optional

Whether to force compression of an empty mask.

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`(*buffer, dtype=float, count=-1, offset=0*)
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.:

```
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

`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

fn : 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]]`. *fn* 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 *fn*.

dtype : data-type, optional

Data-type of the coordinate arrays passed to *fn*. By default, *dtype* is float.

See Also:

`indices`, `meshgrid`

Notes

Keywords other than *shape* and *dtype* are passed to the 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]])
```

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
```

Ones and zeros

ma.empty(shape[,dtype,order])	Return a new array of given shape and type, without initialising entries.
ma.empty_like(a)	Create a new array with the same shape and type as another.
ma.masked_all(shape[,dtype])	Return an empty masked array of the given shape and dtype, where all the data are masked.
ma.masked_all_like(arr)	Return an empty masked array of the same shape and dtype as the array *a*, where all the data are masked.
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 (*shape, dtype=float, order='C'*)

Return a new array of given shape and type, without initialising entries.

Parameters

shape : {tuple of int, 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

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],  #random data
       [  2.13182611e-314,   3.06959433e-309]])
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],  #random data
       [  496041986,    19249760]])
```

empty_like (*a*)

Create a new array with the same shape and type as another.

Parameters

a : ndarray

Returned array will have same shape and type as *a*.

See Also:

`zeros_like`, `ones_like`, `zeros`, `ones`, `empty`

Notes

This function does *not* initialize the returned array; to do that use *zeros_like* or *ones_like* instead.

Examples

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.empty_like(a)
>>> np.empty_like(a)
array([[-1073741821, -1067702173,       65538],    #random data
       [      25670,     23454291,       71800]])
```

masked_all (*shape, dtype=<type 'float'>*)
Return an empty masked array of the given shape and dtype, where all the data are masked.

Parameters
dtype : dtype, optional
Data type of the output.

masked_all_like (*arr*)
Return an empty masked array of the same shape and dtype as the array *a*, where all the data are masked.

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.]])
```

zeros (*shape, dtype=float, order='C'*)
Return a new array of given shape and type, filled with zeros.

Parameters
shape : {tuple of ints, int}
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:

`numpy.zeros_like`
Return an array of zeros with shape and type of input.

`numpy.ones_like`
Return an array of ones with shape and type of input.

`numpy.empty_like`
Return an empty array with shape and type of input.

`numpy.ones`
Return a new array setting values to one.

`numpy.empty`
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')])
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

Inspecting the array

`ma.all`(self[,axis,out])	Check if all of the elements of *a* are true.
`ma.any`(self[,axis,out])	Check if any of the elements of *a* are true.
`ma.count`(a[,axis])	Count the non-masked elements of the array along the given axis.
`ma.count_masked`(arr[,axis])	Count the number of masked elements along the given axis.
`ma.getmask`(a)	Return the mask of a, if any, or nomask.
`ma.getmaskarray`(arr)	Return the mask of arr, if any, or a boolean array of the shape of a, full of False.
`ma.getdata`(a[,subok])	Return the *_data* part of *a* if *a* is a MaskedArray, or *a* itself.
`ma.nonzero`(self)	Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.
`ma.shape`(obj)	Return the shape of an array.
`ma.size`(obj[,axis])	Return the number of elements along a given axis.
`ma.MaskedArray.data`	
`ma.MaskedArray.mask`	Mask
`ma.MaskedArray.recordmask`	
`ma.MaskedArray.all`(self[,axis,out])	Check if all of the elements of *a* are true.
`ma.MaskedArray.any`(self[,axis,out])	Check if any of the elements of *a* are true.
`ma.MaskedArray.count`(self[,axis])	Count the non-masked elements of the array along the given axis.
`ma.MaskedArray.nonzero`(self)	Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.
`ma.shape`(obj)	Return the shape of an array.
`ma.size`(obj[,axis])	Return the number of elements along a given axis.

all(*self, 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

```
>>> 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 (*self, 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 (*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 the non masked elements are counted.
Returns
result : MaskedArray
A masked array where the mask is True where all data are masked. If axis is None, returns either a scalar ot the masked singleton if all values are masked.

count_masked (*arr, axis=None*)
Count the number of masked elements along the given axis.

Parameters
axis : int, optional
Axis along which to count. If None (default), a flattened version of the array is used.

getmask (*a*)
Return the mask of a, if any, or nomask.

To get a full array of booleans of the same shape as a, use getmaskarray.

getmaskarray(*arr*)
Return the mask of arr, if any, or a boolean array of the shape of a, full of False.

getdata(*a, subok=True*)
Return the *_data* part of *a* if *a* is a MaskedArray, or *a* itself.

Parameters
a : array_like
A ndarray or a subclass of.
subok : {True, False}, optional
Whether to force the output to a 'pure' ndarray (False) or to return a subclass of ndarray if approriate (True).

nonzero(*self*)
Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.

There are as many tuples as dimensions of a, each tuple contains 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: `transpose(a.nonzero())`.

The result of this is always a 2d array, with a row for each non-zero element.

shape(*obj*)
Return the shape of an array.

Parameters
a : array_like
Input array.

Returns
shape : tuple
The elements of the tuple give the lengths of the corresponding array dimensions.

See Also:

`alen`

ndarray.shape
array method

Examples

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1,2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1,2),(3,4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

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
>>> np.size(a,0)
2
```

`data`

`mask`
Mask

`recordmask`

`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

```
>>> 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 the non masked elements are counted.
Returns
result : MaskedArray
A masked array where the mask is True where all data are masked. If axis is None, returns either a scalar ot the masked singleton if all values are masked.

nonzero()
Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.

There are as many tuples as dimensions of a, each tuple contains 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: `transpose(a.nonzero())`.

The result of this is always a 2d array, with a row for each non-zero element.

shape(*obj*)
Return the shape of an array.

Parameters
a : array_like
Input array.
Returns
shape : tuple
The elements of the tuple give the lengths of the corresponding array dimensions.

See Also:

`alen`

`ndarray.shape`
array method

Examples

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1,2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1,2),(3,4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

`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
>>> np.size(a,0)
2
```

Manipulating a MaskedArray

Changing the shape

`ma.ravel`(self)	Returns a 1D version of self, as a view.
`ma.reshape`(a,new_shape[,order])	Change the shape of the array a to new_shape.
`ma.resize`(x,new_shape)	Return a new array with the specified shape.
`ma.MaskedArray.flatten`([order])	Collapse an array into one dimension.
`ma.MaskedArray.ravel`(self)	Returns a 1D version of self, as a view.
`ma.MaskedArray.reshape`(self,*s,**kwargs)	Returns a masked array containing the data of a, but with a new shape. The result is a view to the original array; if this is not possible, a ValueError is raised.
`ma.MaskedArray.resize`(self,newshape[,refcheck,order])	Change shape and size of array in-place.

ravel (*self*)
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

```
>>> 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'*)
Change the shape of the array a to new_shape.

resize (*x, new_shape*)
Return a new array with the specified shape.

The total size of the original array can be any size. The new array is filled with repeated copies of a. If a was masked, the new array will be masked, and the new mask will be a repetition of the old one.

flatten (*order='C'*)
Collapse an array 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.

Examples

```
>>> 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

```
>>> 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*)
Returns a masked array containing the data of a, but with a new shape. The result is a view to the original array; if this is not possible, a ValueError is raised.

Parameters
shape : shape tuple or int
The new shape should be compatible with the original shape. If an integer, then the result will be a 1D array of that length.
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 : array
A new view to the array.

Notes

If you want to modify the shape in place, please use `a.shape = s`

Examples

```
>>> 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*)
Change shape and size of array in-place.

Modifying axes

`ma.swapaxes`()	swapaxes a.swapaxes(axis1, axis2)
`ma.transpose`(a[,axes])	Return a view of the array with dimensions permuted according to axes, as a masked 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 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

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*)
Return a view of the array with dimensions permuted according to axes, as a masked array.

If `axes` is None (default), the output view has reversed dimensions compared to the original.

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 (**axes*)
Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

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

Changing the number of dimensions

`ma.atleast_1d`(*arys)	Convert inputs to arrays with at least one dimension.
`ma.atleast_2d`(*arys)	View inputs as arrays with at least two dimensions.
`ma.atleast_3d`(*arys)	View inputs as arrays with at least three dimensions.
`ma.expand_dims`(x,axis)	Expand the shape of the array by including a new axis before the given one.
`ma.squeeze`(a)	Remove single-dimensional entries from the shape of an array.
`ma.MaskedArray.squeeze`()	Remove single-dimensional entries from the shape of *a*.
`ma.column_stack`(tup)	Stack 1-D arrays as columns into a 2-D array
`ma.concatenate`(arrays[,axis])	Concatenate the arrays along the given axis.
`ma.dstack`(tup)	Stack arrays in sequence depth wise (along third axis)
`ma.hstack`(tup)	Stack arrays in sequence horizontally (column wise)
`ma.hsplit`(ary,indices_or_sections)	Split array into multiple sub-arrays horizontally.
`ma.mr_`	Translate slice objects to concatenation along the first axis.
`ma.row_stack`(tup)	Stack arrays vertically.
`ma.vstack`(tup)	Stack arrays vertically.

`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

```
>>> 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

```
>>> numpy.atleast_2d(3.0)
array([[ 3.]])
```

```
>>> x = numpy.arange(3.0)
>>> numpy.atleast_2d(x)
array([[ 0.,  1.,  2.]])
>>> numpy.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 one-dimensional array of shape `N` becomes a view of shape `(1, N, 1)`. An `(M, N)` array becomes a view of shape `(N, M, 1)`.

See Also:

`numpy.atleast_1d`, `numpy.atleast_2d`

Examples

```
>>> numpy.atleast_3d(3.0)
array([[[ 3.]]])
```

```
>>> x = numpy.arange(3.0)
>>> numpy.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = numpy.arange(12.0).reshape(4,3)
>>> numpy.atleast_3d(x).shape
(4, 3, 1)
>>> numpy.atleast_3d(x).base is x
True
```

```
>>> for arr in np.atleast_3d(1, [1, 2], [[1, 2]]): print arr, "\n"
...
[[[1]]]
```

[[[1]

[2]]]

[[[1]

[2]]]

`expand_dims` (*x, axis*)

Expand the shape of the array by including a new axis before the given one.

`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

```
>>> 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(*tup*)

Stack 1-D arrays as columns into a 2-D array

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters
tup : sequence of 1-D or 2-D arrays.
Arrays to stack. All of them must have the same first dimension.

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 the arrays along the given axis.

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

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.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
<BLANKLINE>
       [[2, 3]],
<BLANKLINE>
       [[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 along first axis.

dstack
: Stack along third axis.

concatenate
: Join arrays.

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 (*ary, indices_or_sections*)

Split array into multiple sub-arrays horizontally.

Please refer to the `numpy.split` documentation. *hsplit* is equivalent to `numpy.split` with `axis = 1`.

See Also:

split
Split array into multiple sub-arrays.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> np.hsplit(x, 2)
<BLANKLINE>
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [ 12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [ 10.,  11.],
       [ 14.,  15.]])]
```

```
>>> np.hsplit(x, array([3, 6]))
<BLANKLINE>
[array([[  0.,   1.,   2.],
       [  4.,   5.,   6.],
       [  8.,   9.,  10.],
       [ 12.,  13.,  14.]]),
 array([[  3.],
       [  7.],
       [ 11.],
       [ 15.]]),
 array([], dtype=float64)]
```

mr_()

Translate slice objects to concatenation along the first axis.

Examples

```
>>> 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(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
Split an array into a list of multiple sub-arrays of near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

vsplit
Split array into a list of multiple sub-arrays vertically.

dsplit
Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

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]])
```

vstack(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
Split an array into a list of multiple sub-arrays of near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

vsplit
Split array into a list of multiple sub-arrays vertically.

dsplit
Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

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]])
```

Joining arrays

`ma.column_stack`(tup)	Stack 1-D arrays as columns into a 2-D array
`ma.concatenate`(arrays[,axis])	Concatenate the arrays along the given axis.
`ma.dstack`(tup)	Stack arrays in sequence depth wise (along third axis)
`ma.hstack`(tup)	Stack arrays in sequence horizontally (column wise)
`ma.vstack`(tup)	Stack arrays vertically.

`column_stack`(*tup*)

Stack 1-D arrays as columns into a 2-D array

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters

tup : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

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 the arrays along the given axis.

`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

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.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
<BLANKLINE>
       [[2, 3]],
<BLANKLINE>
       [[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 along first axis.

dstack
Stack along third axis.

concatenate
Join arrays.

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]])
```

vstack(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
Split an array into a list of multiple sub-arrays of near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

vsplit
Split array into a list of multiple sub-arrays vertically.

dsplit
Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

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]])
```

Operations on masks

Creating a mask

`ma.make_mask`(m[,copy,shrink,flag,...])	Return m as a mask, creating a copy if necessary or requested.
`ma.make_mask_none`(newshape[,dtype])	Return a mask of shape s, filled with False.
`ma.mask_or`(m1,m2[,copy,shrink])	Return the combination of two masks m1 and m2.
`ma.make_mask_descr`(ndtype)	Constructs a dtype description list from a given dtype. Each field is set to a bool.

make_mask (*m, copy=False, shrink=True, flag=None, dtype=<type 'numpy.bool_'>*)

Return m as a mask, creating a copy if necessary or requested.

The function can accept any sequence of integers or nomask. Does not check that contents must be 0s and 1s.

Parameters

m : array_like

Potential mask.

copy : bool

Whether to return a copy of m (True) or m itself (False).

shrink : bool

Whether to shrink m to nomask if all its values are False.

dtype : dtype

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.

make_mask_none (*newshape, dtype=None*)

Return a mask of shape s, filled with False.

Parameters

news : tuple

A tuple indicating the shape of the final mask.

dtype: {None, dtype}, optional :

If None, use MaskType. Otherwise, use a new datatype with the same fields as *dtype* with boolean type.

`mask_or`(*m1, m2, copy=False, shrink=True*)

Return the combination of two masks m1 and m2.

The masks are combined with the *logical_or* operator, treating nomask as False. The result may equal m1 or m2 if the other is nomask.

Parameters

m1 : array_like

First mask.

m2 : array_like

Second mask

copy : {False, True}, optional

Whether to return a copy.

shrink : {True, False}, optional

Whether to shrink m to nomask if all its values are False.

Raises

ValueError :

If m1 and m2 have different flexible dtypes.

`make_mask_descr`(*ndtype*)

Constructs a dtype description list from a given dtype. Each field is set to a bool.

Accessing a mask

`ma.getmask`(a)	Return the mask of a, if any, or nomask.
`ma.getmaskarray`(arr)	Return the mask of arr, if any, or a boolean array of the shape of a, full of False.
`ma.masked_array.mask`	Mask

`getmask`(*a*)

Return the mask of a, if any, or nomask.

To get a full array of booleans of the same shape as a, use getmaskarray.

`getmaskarray`(*arr*)

Return the mask of arr, if any, or a boolean array of the shape of a, full of False.

`mask`

Mask

Finding masked data

`ma.flatnotmasked_contiguous`(a)	Find contiguous unmasked data in a flattened masked array.
`ma.flatnotmasked_edges`(a)	Find the indices of the first and last not masked values in a 1D masked array. If all values are masked, returns None.
`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 not masked values along the given axis in a masked array.

flatnotmasked_contiguous(*a*)
Find contiguous unmasked data in a flattened masked array.

Return a sorted sequence of slices (start index, end index).

flatnotmasked_edges(*a*)
Find the indices of the first and last not masked values in a 1D masked array. If all values are masked, returns None.

notmasked_contiguous(*a, axis=None*)
Find contiguous unmasked data in a masked array along the given axis.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns
A sorted sequence of slices (start index, end index). :

Notes

Only accepts 2D arrays at most.

notmasked_edges(*a, axis=None*)
Find the indices of the first and last not masked values along the given axis in a masked array.

If all values are masked, return None. Otherwise, return a list of 2 tuples, corresponding to the indices of the first and last unmasked values respectively.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

Modifying a mask

`ma.mask_cols`(a[,axis])	Mask whole columns of a 2D array that contain masked values.
`ma.mask_or`(m1,m2[,copy,shrink])	Return the combination of two masks m1 and m2.
`ma.mask_rowcols`(a[,axis])	Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected with the *axis* parameter.
`ma.mask_rows`(a[,axis])	Mask whole rows of a 2D array that contain masked values.
`ma.harden_mask`()	harden_mask(self) Force the mask to hard.
`ma.soften_mask`()	soften_mask(self) Force the mask to soft.
`ma.MaskedArray.harden_mask`(self)	Force the mask to hard.
`ma.MaskedArray.soften_mask`(self)	Force the mask to soft.
`ma.MaskedArray.shrink_mask`(self)	Reduce a mask to nomask when possible.
`ma.MaskedArray.unshare_mask`(self)	Copy the mask and set the sharedmask flag to False.

mask_cols(*a, axis=None*)
Mask whole columns of a 2D array that contain masked values.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

`mask_or`(*m1, m2, copy=False, shrink=True*)
Return the combination of two masks m1 and m2.

The masks are combined with the *logical_or* operator, treating nomask as False. The result may equal m1 or m2 if the other is nomask.

Parameters
m1 : array_like
First mask.
m2 : array_like
Second mask
copy : {False, True}, optional
Whether to return a copy.
shrink : {True, False}, optional
Whether to shrink m to nomask if all its values are False.

Raises
ValueError :
If m1 and m2 have different flexible dtypes.

`mask_rowcols`(*a, axis=None*)
Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected with 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
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns
a *pure* ndarray. :

`mask_rows`(*a, axis=None*)
Mask whole rows of a 2D array that contain masked values.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

`harden_mask`()
harden_mask(self) Force the mask to hard.

`soften_mask`()
soften_mask(self) Force the mask to soft.

`harden_mask`()
Force the mask to hard.

soften_mask()
Force the mask to soft.

shrink_mask()
Reduce a mask to nomask when possible.

unshare_mask()
Copy the mask and set the sharedmask flag to False.

Conversion operations

> to a masked array

`ma.asarray`(a[,dtype,order])	Convert the input *a* to a masked array of the given datatype.
`ma.asanyarray`(a[,dtype])	Convert the input *a* to a masked array of the given datatype. If *a* is a subclass of MaskedArray, its class is conserved.
`ma.fix_invalid`(a[,mask,copy,fill_value])	Return (a copy of) *a* where invalid data (nan/inf) are masked and replaced by *fill_value*.
`ma.masked_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x == value)`.
`ma.masked_greater`(x,value[,copy])	Return the array *x* masked where `(x > value)`. Any value of mask already masked is kept masked.
`ma.masked_greater_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x >= value)`.
`ma.masked_inside`(x,v1,v2[,copy])	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.
`ma.masked_invalid`(a[,copy])	Mask the array for invalid values (NaNs or infs). Any preexisting mask is conserved.
`ma.masked_less`(x,value[,copy])	Shortcut to masked_where, with condition `(x < value)`.
`ma.masked_less_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x <= value)`.
`ma.masked_not_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x != value)`.
`ma.masked_object`(x,value[,copy,shrink])	Mask the array x where the data are exactly equal to value.
`ma.masked_outside`(x,v1,v2[,copy])	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.
`ma.masked_values`(x,value[,rtol,atol,copy,...])	Mask the array x where the data are approximately equal in value, i.e. `(abs(x - value) <= atol+rtol*abs(value))`
`ma.masked_where`(condition,a[,copy])	Return a as an array masked where `condition` is `True`. Masked values of `a` or `condition` are kept.

asarray(*a, dtype=None, order=None*)
Convert the input *a* to a masked array of the given datatype.

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 ('FORTRAN') memory representation. Defaults to 'C'.

Returns

out : ndarray

MaskedArray interpretation of *a*. No copy is performed if the input is already an ndarray. If *a* is a subclass of MaskedArray, a base class MaskedArray is returned.

`asanyarray`(*a, dtype=None*)

Convert the input *a* to a masked array of the given datatype. If *a* is a subclass of MaskedArray, its class is conserved.

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 ('FORTRAN') memory representation. Defaults to 'C'.

Returns

out : ndarray

MaskedArray interpretation of *a*. No copy is performed if the input is already an ndarray.

`fix_invalid`(*a, mask=False, copy=True, fill_value=None*)

Return (a copy of) *a* where invalid data (nan/inf) are masked and replaced by *fill_value*.

Note that a copy is performed by default (just in case...).

Parameters

a : array_like

A (subclass of) ndarray.

copy : bool

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

fill_value : {var}, optional

Value used for fixing invalid data. If not given, the output of get_fill_value(a) is used instead.

Returns

b : MaskedArray

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

Shortcut to masked_where, with condition `(x == value)`.

See Also:

`masked_where`
base function

`masked_values`
equivalent function for floats.

`masked_greater` (*x, value, copy=True*)
Return the array *x* masked where `(x > value)`. Any value of mask already masked is kept masked.

`masked_greater_equal` (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x >= value)`.

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

Notes

The array x is prefilled with its filling value.

`masked_invalid` (*a, copy=True*)
Mask the array for invalid values (NaNs or infs). Any preexisting mask is conserved.

`masked_less` (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x < value)`.

`masked_less_equal` (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x <= value)`.

`masked_not_equal` (*x, value, copy=True*)
Shortcut to masked_where, with condition `(x != value)`.

`masked_object` (*x, value, copy=True, shrink=True*)
Mask the array *x* where the data are exactly equal to value.

This function is suitable only for object arrays: for floating point, please use **'masked_values'_** instead.

Parameters
x : array_like
Array to mask
value : var
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

`masked_outside` (*x, v1, v2, copy=True*)
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.

Notes

The array x is prefilled with its filling value.

`masked_values` (*x, value, rtol=1.0000000000000001e-05, atol=1e-08, copy=True, shrink=True*)
Mask the array x where the data are approximately equal in value, i.e. `(abs(x - value) <= atol+rtol*abs(value))`

Suitable only for floating points. For integers, please use `masked_equal`. The mask is set to `nomask` if posible.

Parameters
x : array_like

Array to fill.

value : float

Masking value.

rtol : {float}, optional

Tolerance parameter.

atol : {float}, optional

Tolerance parameter (1e-8).

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

masked_where(*condition, a, copy=True*)

Return `a` as an array masked where `condition` is `True`. Masked values of `a` or `condition` are kept.

Parameters

condition : array_like

Masking condition.

a : array_like

Array to mask.

copy : bool

Whether to return a copy of `a` (True) or modify `a` in place (False).

> to a ndarray

`ma.compress_cols`(a)	Suppress whole columns of a 2D array that contain masked values.
`ma.compress_rowcols`(x[,axis])	Suppress the rows and/or columns of a 2D array that contain masked values.
`ma.compress_rows`(a)	Suppress whole rows of a 2D array that contain masked values.
`ma.compressed`(x)	Return a 1-D array of all the non-masked data.
`ma.filled`(a[,fill_value])	Return *a* as an array where masked data have been replaced by *value*.
`ma.MaskedArray.compressed`(self)	Return a 1-D array of all the non-masked data.
`ma.MaskedArray.filled`(self[,fill_value])	Return a copy of self, where masked values are filled with *fill_value*.

compress_cols(*a*)

Suppress whole columns of a 2D array that contain masked values.

compress_rowcols(*x, axis=None*)

Suppress the rows and/or columns of a 2D array that contain masked values.

The suppression behavior is selected with the *axis* parameter.

- If axis is None, 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. If None, applies to a flattened version of the array.

Returns

compressed_array : an ndarray.

compress_rows(*a*)

Suppress whole rows of a 2D array that contain masked values.

compressed(*x*)

Return a 1-D array of all the non-masked data.

See Also:

MaskedArray.compressed
equivalent method

filled(*a, fill_value=None*)

Return *a* as an array where masked data have been replaced by *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 : {var}, optional

Filling value. If None, the output of `get_fill_value(a)` is used instead.

Returns

a : array_like

compressed()

Return a 1-D array of all the non-masked data.

Returns

data : ndarray.

A new ndarray holding the non-masked data is returned.

Notes

- The result is NOT a MaskedArray !

Examples

```
>>> x = array(arange(5), mask=[0]+[1]*4)
>>> print x.compressed()
[0]
>>> print type(x.compressed())
<type 'numpy.ndarray'>
```

filled(*fill_value=None*)

Return a copy of self, where masked values are filled with *fill_value*.

If *fill_value* is None, *self.fill_value* is used instead.

Notes

- Subclassing is preserved
- The result is NOT a MaskedArray !

Examples

```
>>> 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'>
```

> to another object

`ma.MaskedArray.tofile`(self,fid[,sep,format])	
`ma.MaskedArray.tolist`(self[,fill_value])	Copy the data portion of the array to a hierarchical python list and returns that list.
`ma.MaskedArray.torecords`(self)	Transforms a MaskedArray into a flexible-type array with two fields:
`ma.MaskedArray.tostring`(self[,fill_value,order])	Return the array data as a Python string containing the raw bytes in the array. The array is filled beforehand.

tofile (*fid, sep='', format='%s'*)

tolist (*fill_value=None*)
Copy the data portion of the array to a hierarchical python list and returns that 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`.

torecords ()
Transforms a MaskedArray into a flexible-type array with two fields:

- the `_data` field stores the `_data` part of the array;
- the `_mask` field stores the `_mask` part of the array;

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

```
>>> 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.torecords()
[[(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 a copy of array data as a Python string containing the raw bytes in the array. The array is filled beforehand.

Parameters
fill_value : {var}, optional
Value used to fill in the masked values. If None, uses self.fill_value instead.
order : {string}
Order of the data item in the copy {'C','F','A'}. 'C' – C order (row major) 'Fortran' – Fortran order (column major) 'Any' – Current order of array. None – Same as "Any"

Notes

As for method:*ndarray.tostring*, information about the shape, dtype..., but also fill_value will be lost.

Pickling and unpickling

`ma.dump`(a,F)	Pickle the MaskedArray *a* to the file *F*. *F* can either be the handle of an exiting file, or a string representing a file name.
`ma.dumps`(a)	Return a string corresponding to the pickling of the MaskedArray.
`ma.load`(F)	Wrapper around `cPickle.load` which accepts either a file-like object or a filename.
`ma.loads`(strg)	Load a pickle from the current string.

dump(*a, F*)
Pickle the MaskedArray *a* to the file *F*. *F* can either be the handle of an exiting file, or a string representing a file name.

dumps(*a*)
Return a string corresponding to the pickling of the MaskedArray.

load(*F*)
Wrapper around `cPickle.load` which accepts either a file-like object or a filename.

loads(*strg*)
Load a pickle from the current string.

Filling a masked array

ma.common_fill_value(a,b)	Return the common filling value of a and b, if any. If a and b have different filling values, returns None.
ma.default_fill_value(obj)	Calculate the default fill value for the argument object.
ma.maximum_fill_value(obj)	Calculate the default fill value suitable for taking the maximum of obj.
ma.maximum_fill_value(obj)	Calculate the default fill value suitable for taking the maximum of obj.
ma.set_fill_value(a,fill_value)	Set the filling value of a, if a is a masked array. Otherwise, do nothing.
ma.MaskedArray.get_fill_value(self)	Return the filling value.
ma.MaskedArray.set_fill_value(self[, value])	Set the filling value to value.
ma.MaskedArray.fill_value	Filling value.

common_fill_value(*a, b*)
Return the common filling value of a and b, if any. If a and b have different filling values, returns None.

default_fill_value(*obj*)
Calculate the default fill value for the argument object.

maximum_fill_value(*obj*)
Calculate the default fill value suitable for taking the maximum of `obj`.

maximum_fill_value(*obj*)
Calculate the default fill value suitable for taking the maximum of `obj`.

set_fill_value(*a, fill_value*)
Set the filling value of a, if a is a masked array. Otherwise, do nothing.

Parameters
a : ndarray
Input array
fill_value : var
Filling value. A consistency test is performed to make sure the value is compatible with the dtype of a.

Returns
None :

get_fill_value()
Return the filling value.

set_fill_value(*value=None*)
Set the filling value to value.

If value is None, use a default based on the data type.

fill_value
Filling value.

Masked arrays arithmetics

Arithmetics

`ma.anom`(self[,axis,dtype])	Return the anomalies (deviations from the average) along the given axis.
`ma.anomalies`(self[,axis,dtype])	Return the anomalies (deviations from the average) along the given axis.
`ma.average`(a[,axis,weights,returned])	Average the array over the given axis.
`ma.conjugate`(x[,out])	Return the complex conjugate, element-wise.
`ma.corrcoef`(x[,y,rowvar,bias,...])	The correlation coefficients formed from the array x, where the rows are the observations, and the columns are variables.
`ma.cov`(x[,y,rowvar,bias,...])	Estimates the covariance matrix.
`ma.cumsum`(self[,axis,dtype,out])	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,dtype,out])	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`(self[,axis,dtype,out])	Returns the average of the array elements along given axis. Refer to `numpy.mean` for full documentation.
`ma.median`(a[,axis,out,overwrite_input])	Compute the median along the specified axis.
`ma.power`(a,b[,third])	Computes a**b elementwise.
`ma.prod`(self[,axis,dtype,out])	Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.
`ma.std`(self[,axis,dtype,out,...])	Compute the standard deviation along the specified axis.
`ma.sum`(self[,axis,dtype,out])	Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
`ma.var`(self[,axis,dtype,out,...])	Compute the variance along the specified axis.
`ma.MaskedArray.anom`(self[,axis,dtype])	Return the anomalies (deviations from the average) along the given axis.
`ma.MaskedArray.cumprod`(self[,axis,dtype,out])	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.cumsum`(self[,axis,dtype,out])	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.mean`(self[,axis,dtype,out])	Returns the average of the array elements along given axis. Refer to `numpy.mean` for full documentation.
`ma.MaskedArray.prod`(self[,axis,dtype,out])	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,dtype,out,...])	Compute the standard deviation along the specified axis.
`ma.MaskedArray.sum`(sel	Return the sum of the array elements over the given axis. Masked elements are set

anom (*self, axis=None, dtype=None*)

Return the anomalies (deviations from the average) along the given axis.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

dtype : {dtype}, optional

Datatype for the intermediary computation. If not given, the current dtype is used instead.

anomalies (*self, axis=None, dtype=None*)

Return the anomalies (deviations from the average) along the given axis.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

dtype : {dtype}, optional

Datatype for the intermediary computation. If not given, the current dtype is used instead.

average (*a, axis=None, weights=None, returned=False*)

Average the array over the given axis.

Parameters

axis : {None,int}, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

weights : {None, sequence}, optional

Sequence of weights. The weights must have the shape of a, or be 1D with length the size of a along the given axis. If no weights are given, weights are assumed to be 1.

returned : {False, True}, optional

Flag indicating whether a tuple (result, sum of weights/counts) should be returned as output (True), or just the result (False).

conjugate (*x, [out]*)

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

```
>>> np.conjugate(1+2j)
(1-2j)
```

corrcoef (*x, y=None, rowvar=True, bias=False, allow_masked=True*)

The correlation coefficients formed from the array x, where the rows are the observations, and the columns are variables.

corrcoef(x,y) where x and y are 1d arrays is the same as corrcoef(transpose([x,y]))

Parameters

x : ndarray

Input data. If x is a 1D array, returns the variance. If x is a 2D array, returns the covariance matrix.

y : {None, ndarray} optional

Optional set of variables.

rowvar : {False, True} optional

If True, then each row is a variable with observations in columns. If False, each column is a variable and the observations are in the rows.

bias : {False, True} optional

Whether to use a biased (True) or unbiased (False) estimate of the covariance. If True, then the normalization is by N, the number of non-missing observations. Otherwise, the normalization is by (N-1).

allow_masked : {True, False} 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:

`cov`

`cov` (*x, y=None, rowvar=True, bias=False, allow_masked=True*)

Estimates the covariance matrix.

Normalization is by (N-1) where N is the number of observations (unbiased estimate). If bias is True then normalization is by N.

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

Input data. If x is a 1D array, returns the variance. If x is a 2D array, returns the covariance matrix.

y : array_like, optional

Optional set of variables.

rowvar : {False, True} optional

If rowvar is true, then each row is a variable with observations in columns. If rowvar is False, each column is a variable and the observations are in the rows.

bias : {False, True} optional

Whether to use a biased (True) or unbiased (False) estimate of the covariance. If bias is True, then the normalization is by N, the number of observations. Otherwise, the normalization is by (N-1).

allow_masked : {True, False} 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.

`cumsum` (*self, 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.

Warning: The mask is lost if out is not a valid `MaskedArray` !

Notes

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]
```

cumprod(*self, 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

Arithmetic is modular when using integer types, and no error is raised on overflow.

mean (*self, axis=None, dtype=None, out=None*)

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

See Also:

`numpy.mean`
equivalent function'

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 (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 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 is returned unless out is specified, in which case a reference to out is returned. Return datatype is float64 for ints and floats smaller than float64, or the input datatype 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.

power (*a, b, third=None*)

Computes a**b elementwise.

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.

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.])
```

`std`(*self, 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 (biased estimate).

Returns

standard_deviation : {ndarray, scalar}; 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:

numpy.var
Variance

numpy.mean
Average

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `var = sqrt(mean(abs(x - x.mean())**2))`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, 0)
array([ 1.,  1.])
>>> np.std(a, 1)
array([ 0.5,  0.5])
```

sum(*self, 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

```
>>> 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 (*self, 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

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of 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.

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])
```

anom(*axis=None, dtype=None*)
Return the anomalies (deviations from the average) along the given axis.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.
dtype : {dtype}, optional
Datatype for the intermediary computation. If not given, the current dtype is used instead.

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

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.

Warning: The mask is lost if out is not a valid `MaskedArray` !

Notes

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 along given axis. Refer to `numpy.mean` for full documentation.

See Also:

numpy.mean
equivalent function'

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.])
```

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 (biased estimate).

Returns

standard_deviation : {ndarray, scalar}; 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:

numpy.var
Variance

numpy.mean
Average

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `var = sqrt(mean(abs(x - x.mean())**2))`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, 0)
array([ 1.,  1.])
>>> np.std(a, 1)
array([ 0.5,  0.5])
```

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

```
>>> 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.

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

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of 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.

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])
```

Minimum/maximum

`ma.argmax`(a[,axis,fill_value])	Function version of the eponymous method.
`ma.argmin`(a[,axis,fill_value])	Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.
`ma.max`(obj[,axis,out,fill_value])	Return the maximum along a given axis.
`ma.min`(obj[,axis,out,fill_value])	Return the minimum along a given axis.
`ma.ptp`(obj[,axis,out,fill_value])	Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).
`ma.MaskedArray.argmax`(self[,axis,fill_value,out])	Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.
`ma.MaskedArray.argmin`(self[,axis,fill_value,out])	Return array of indices to the minimum values along the given axis.
`ma.MaskedArray.max`(self[,axis,out,fill_value])	Return the maximum along a given axis.
`ma.MaskedArray.min`(self[,axis,out,fill_value])	Return the minimum along a given axis.
`ma.MaskedArray.ptp`(self[,axis,out,fill_value])	Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

argmax (*a, axis=None, fill_value=None*)
Function version of the eponymous method.

argmin (*a, axis=None, fill_value=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])
```

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

```
>>> 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]
```

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. If not given, 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.

`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 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:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```
>>> 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:

```
>>> 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. If not given, 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.

`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:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```
>>> 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:

```
>>> 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')])
```

Algebra

`ma.diag`(v[,k])	Extract a diagonal or construct a diagonal array.
`ma.dot`(a,b[,strict])	Return the dot product of two 2D masked arrays a and b.
`ma.identity`(n[,dtype])	Return the identity array.
`ma.inner`(a,b)	Inner product of two arrays.
`ma.innerproduct`(a,b)	Inner product of two arrays.
`ma.outer`(a,b)	Returns the outer product of two vectors.
`ma.outerproduct`(a,b)	Returns the outer product of two vectors.
`ma.trace`(self[,offset,axis1,axis2,...])	Return the sum along diagonals of the array.
`ma.transpose`(a[,axes])	Return a view of the array with dimensions permuted according to axes, as a masked array.
`ma.MaskedArray.trace`([offset,axis1,axis2,...])	Return the sum along diagonals of the array.
`ma.MaskedArray.transpose`(*axes)	Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

diag (*v, k=0*)

Extract a diagonal or construct a diagonal array.

Parameters

v : array_like

If *v* is a 2-dimensional array, return a copy of its *k*-th diagonal. If *v* is a 1-dimensional array, return a 2-dimensional array with *v* on the *k*-th diagonal.

k : int, optional

Diagonal in question. The defaults is 0.

Examples

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

dot (*a, b, strict=False*)

Return the dot product of two 2D masked arrays a and b.

Like the generic numpy equivalent, the product sum is over the last dimension of a and the second-to-last dimension of b. If strict is True, masked values are propagated: if a masked value appears in a row or column, the whole row or column is considered masked.

Parameters

strict : {boolean}

Whether masked data are propagated (True) or set to 0 for the computation.

Notes

The first argument is not conjugated.

`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.]])
```

`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:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

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:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

outer (*a, b*)

Returns the outer product of two vectors.

Given two vectors, `[a0, a1, ..., aM]` and `[b0, b1, ..., bN]`, the outer product becomes:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

Parameters

a : array_like, shaped (M,)

First input vector. If either of the input vectors are not 1-dimensional, they are flattened.

b : array_like, shaped (N,)

Second input vector.

Returns

out : ndarray, shaped (M, N)

`out[i, j] = a[i] * b[j]`

Notes

Masked values are replaced by 0.

Examples

```
>>> 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*)

Returns the outer product of two vectors.

Given two vectors, `[a0, a1, ..., aM]` and `[b0, b1, ..., bN]`, the outer product becomes:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

Parameters

a : array_like, shaped (M,)

First input vector. If either of the input vectors are not 1-dimensional, they are flattened.

b : array_like, shaped (N,)

Second input vector.

Returns

out : ndarray, shaped (M, N)

`out[i, j] = a[i] * b[j]`

Notes

Masked values are replaced by 0.

Examples

```
>>> 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` (*self, offset=0, axis1=0, axis2=1, dtype=None, out=None) a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

`numpy.trace`
equivalent function

`transpose` (*a, axes=None*)

Return a view of the array with dimensions permuted according to axes, as a masked array.

If `axes` is None (default), the output view has reversed dimensions compared to the original.

`trace` (*offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

`numpy.trace`
equivalent function

`transpose` (**axes*)

Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

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

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`.

Parameters

x : array_like

Input array.

N : int, optional

Order of (number of columns in) the output.

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.

Examples

```
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

polyfit (*x, y, deg, rcond=None, full=False*)

Least squares polynomial fit.

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* to points *(x, y)*. Returns a vector of coefficients *p* that minimises the squared error.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int

Degree of the fitting polynomial

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

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

Wikipedia, “Polynomial interpolation”, http://en.wikipedia.org/wiki/Polynomial_interpolation

Examples

```
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254])
```

It is convenient to use *poly1d* objects for dealing with polynomials:

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
>>> plt.show()
```

Clipping and rounding

`ma.around`()	Round an array to the given number of decimals.
`ma.clip`(a,a_min,a_max[,out])	Clip (limit) the values in an array.
`ma.round`(a[,decimals,out])	Return a copy of a, rounded to 'decimals' places.
`ma.MaskedArray.clip`(a_min,a_max[,out])	Return an array whose values are limited to `[a_min, a_max]`.
`ma.MaskedArray.round`([decimals,out])	Return an array rounded a to the given number of decimals.

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*.

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, [3,4,1,1,1,4,4,4,4,4], 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(*a_min, a_max, out=None*)

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,...])	Returns True if two arrays are element-wise equal within a tolerance.
`ma.apply_along_axis`(func1d,axis,arr,...)	Apply function to 1-D slices along the given axis.
`ma.arange`([start,]stop[,step,...])	Return evenly spaced values within a given interval.
`ma.choose`(a,choices[,out,mode])	Use an index array to construct a new array from a set of choices.
`ma.ediff1d`(arr[,to_end,to_begin])	Computes 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])	Returns a (subclass of) masked array, shaped like condition, where the elements are x when condition is True, and y otherwise. If neither x nor y are given, returns a tuple of indices where condition is True (a la condition.nonzero()).

`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.

`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.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * *b*) and the absolute difference (*atol*) are added together to compare against the absolute difference between *a* and *b*.

Parameters

a, b : array_like
Input arrays to compare.

fill_value : boolean, optional
Whether masked values in a or b are considered equal (True) or not (False).

rtol : Relative tolerance
The relative difference is equal to *rtol* * *b*.

atol : Absolute tolerance
The absolute difference is equal to *atol*.

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`, `alltrue`, `sometrue`

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.

`apply_along_axis` (*func1d, axis, arr, *args, **kwargs*)

Apply function to 1-D slices along the given axis.

Execute *func1d(a[i],*args)* where *func1d* takes 1-D arrays, *a* is the input array, and *i* is an integer that varies in order to apply the function along the given axis for each 1-D subarray in *a*.

Parameters

func1d : function

This function should be able to take 1-D arrays. It is applied to 1-D slices of *a* along the specified axis.

axis : integer

Axis along which *func1d* is applied.

a : ndarray

Input array.

args : any

Additional arguments to *func1d*.

Returns

out : ndarray

The output array. The shape of *out* is identical to the shape of *a*, except along the *axis* dimension, whose length is equal to the size of the return value of *func1d*.

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.])
```

`arange` (*[start], stop, [step], dtype=None*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in range 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

```
>>> 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`(*a, 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 : int array

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

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

Computes the differences between consecutive elements of an array.

This function is the equivalent of `numpy.ediff1d` that takes masked values into account.

Returns

output : MaskedArray

See Also:

`numpy.eddif1d`
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 : 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]])
```

where (*condition | x, y*)

Returns a (subclass of) masked array, shaped like condition, where the elements are x when condition is True, and y otherwise. If neither x nor y are given, returns a tuple of indices where condition is True (a la condition.nonzero()).

Parameters

condition : {var}

The condition to meet. Must be convertible to an integer array.

x : {var}, optional

Values of the output when the condition is met

y : {var}, optional

Values of the output when the condition is not met.

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`. **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.

`__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:

`t`	Bit field (following integer gives the number of bits in the bit field).
`b`	Boolean (integer type where all values are only True or False)
`i`	Integer
`u`	Unsigned integer
`f`	Floating point
`c`	Complex floating point
`O`	Object (i.e. the memory contains a pointer to `PyObject`)
`S`	String (fixed-length sequence of char)
`U`	Unicode (fixed-length sequence of `Py_UNICODE`)
`V`	Other (void * – each item is a fixed-size chunk of memory)

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 appropiately, 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 `PyCObject` whose `voidptr` member contains a pointer to a filled `PyArrayInterface` structure. Memory for the structure is dynamically created and the `PyCObject` 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.

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`.

CHAPTER

TWO

UNIVERSAL FUNCTIONS (`UFUNC`)

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 shapes in that dimension.
3. An input can be used in the calculation if it's shape in a particular dimension either matches the output shape 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 when otherwise needed (the *stride* will be 0 for that dimension).

Broadcasting is used throughout NumPy to decide how to handle non equally-shaped arrays; for example all arithmetic operators (`+`, `-`, `*`, ...) 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_wrap__` method 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.

The ufuncs can also all take output arguments. The output will be cast if necessary to the provided output array. 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_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 the internal buffers is settable on a per-thread basis. There can be up to $2(n_{\mathrm{inputs}} + n_{\mathrm{outputs}})$ buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of the 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 incorrect 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

`setbufsize`(size)	Set the size of the buffer used in ufuncs.

`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

`seterr`([all,divide,over,...])	Set how floating-point errors are handled.
`seterrcall`(func)	Set the floating-point error callback function or log object.

`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'}, 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.

The default is not to change the current behavior.

divide : {'ignore', 'warn', 'raise', 'call'}, optional

Treatment for division by zero.

over : {'ignore', 'warn', 'raise', 'call'}, optional

Treatment for floating-point overflow.

under : {'ignore', 'warn', 'raise', 'call'}, optional

Treatment for floating-point underflow.

invalid : {'ignore', 'warn', 'raise', 'call'}, 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

Set mode:

```
>>> seterr(over='raise') # doctest: +SKIP
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore',
 'under': 'ignore'}
```

```
>>> old_settings = seterr(all='warn', over='raise') # doctest: +SKIP
>>> int16(32000) * int16(3) # doctest: +SKIP
Traceback (most recent call last):
      File "<stdin>", line 1, in ?
FloatingPointError: overflow encountered in short_scalars
>>> seterr(all='ignore') # doctest: +SKIP
{'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

log_func_or_obj : 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 invalid over under invalid]
```

In other words, `flags = divide + 2*over + 4*under + 8*invalid`.
If an object is provided, it's write method should take one argument, a string.

Returns

h : callable or log instance

The old error handler.

Examples

Callback upon error:

```
>>> def err_handler(type, flag):
    print "Floating point error (%s), with flag %s" % (type, flag)
...
```

```
>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')
```

```
>>> np.array([1,2,3])/0.0
Floating point error (divide by zero), with flag 1
array([ Inf,  Inf,  Inf])
```

```
>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)
```

Log error message:

```
>>> class Log(object):
        def write(self, msg):
            print "LOG: %s" % msg
...
```

```
>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')

>>> np.array([1,2,3])/0.0
LOG: Warning: divide by zero encountered in divide

>>> 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 that 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 with types cast. 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)`. Figure shows the results of this call for my 32-bit system on the 21 internally supported types. You can generate this table for your system with code shown in that Figure.

Figure

Code segment showing the can cast safely table for a 32-bit system.

```
>>> 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 0 1 1 1 1 1 1 1 1 1 1
h 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
i 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
l 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
q 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
p 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
B 0 0 1 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 0 1 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 1
L 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1
Q 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1
P 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1
f 0 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 0 1 1 1 1 1 1
g 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1
F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
U 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
V 0 0 0 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 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 hierachy 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 likely not be used by most 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 specify a specific signature for a 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 key-word argument lets you by-pass that search and choose a loop you want. A list of available signatures is available in 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 call-back function. Normally, these values are looked-up in a thread-specific dictionary. Passing them here bypasses that look-up and uses the low-level specification provided for the error-mode. This may be useful as an optimization for calculations requiring lots of ufuncs 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.

__doc__	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 doc string is provided at creation time and stored with the ufunc.
__name__	The name of the ufunc.

`ufunc.nin`	number of inputs
`ufunc.nout`	number of outputs
`ufunc.nargs`	number of arguments
`ufunc.ntypes`	number of types
`ufunc.types`	return a list with types grouped input->output
`ufunc.identity`	identity value

`nin`
number of inputs

`nout`
number of outputs

`nargs`
number of arguments

`ntypes`
number of types

`types`
return a list with types grouped input->output

`identity`
identity value

2.6.3 Methods

All ufuncs have 4 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 >= 1. The *axis* keyword specifies which axis of the array the reduction will take place over 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 that the reduction takes place over (and therefore the type of the output). Thus, you can ensure that the output is a data type with large-enough precision 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.

`ufunc.reduce`(array[,axis,dtype,out])	Reduce applies the operator to all elements of the array.
`ufunc.accumulate`(array[,axis,dtype,out])	Accumulate the result of applying the operator to all elements.
`ufunc.reduceat`(self,array,indices[,axis,dtype,out])	Reduceat performs a reduce with specified slices over an axis.
`ufunc.outer`(A,B)	Compute the result of applying op to all pairs (a,b)

reduce (*array, axis=0, dtype=None, out=None*)

Reduce applies the operator to all elements of the array.

For a one-dimensional array, reduce produces results equivalent to:

```
r = op.identity
for i in xrange(len(A)):
  r = op(r,A[i])
return r
```

For example, add.reduce() is equivalent to sum().

Parameters

array : array_like

The array to act on.

axis : integer, 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 : array_like, 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 equal to out.

Examples

```
>>> np.multiply.reduce([2,3,5])
30
```

accumulate (*array, axis=None, dtype=None, out=None*)

Accumulate the result of applying the operator to all elements.

For a one-dimensional array, accumulate produces results equivalent to:

```
r = np.empty(len(A))
t = op.identity
for i in xrange(len(A)):
   t = op(t,A[i])
   r[i] = t
return r
```

For example, add.accumulate() is equivalent to cumsum().

Parameters

array : array_like

The array to act on.

axis : int, optional

The axis along which to apply the accumulation.

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 accumulated values. If *out* was supplied, *r* is equal to *out*.

Examples

```
>>> np.multiply.accumulate([2,3,5])
array([2,6,30])
```

`reduceat` (*self, array, indices, axis=None, dtype=None, out=None*)

Reduceat performs a reduce with specified slices over an axis.

Computes op.reduce(*array[indices[i]:indices[i+1]]*) for i=0..end with an implicit *indices[i+1]* = len(*array*) assumed when i = end - 1.

If *indices[i]* >= *indices[i + 1]* then the result is *array[indices[i]]* for that value.

The function op.accumulate(*array*) is the same as op.reduceat(*array*, *indices*)[::2] where *indices* is range(len(*array*)-1) with a zero placed in every other sample: *indices* = zeros(len(*array*)*2 - 1) *indices[1::2]* = range(1, len(*array*))

The output shape is based on the size of *indices*.

Parameters

array : array_like

The array to act on.

indices : array_like

Paired indices specifying slices to reduce.

axis : int, optional

The axis along which to apply the reduceat.

dtype : data-type-code, optional

The type used to represent the intermediate results. Defaults to the data type of the output array if this is provided, or the data type of the input array if no output array is provided.

out : ndarray, optional

A location into which the result is stored. If not provided a freshly-allocated array is returned.

Returns

r : array

The reduced values. If *out* was supplied, *r* is equal to *out*.

Examples

To take the running sum of four successive values:

```
>>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]
array([ 6, 10, 14, 18])
```

`outer`(*A*, *B*)

Compute the result of applying op to all pairs (a,b)

op.outer(A,B) is equivalent to op(A[:,:,...,:,newaxis,...,newaxis]*B[newaxis,...,newaxis,:,...,:]) where A has B.ndim new axes appended and B has A.ndim new axes prepended.

For A and B one-dimensional, this is equivalent to

```
r = empty(len(A),len(B))
for i in xrange(len(A)):
    for j in xrange(len(B)):
        r[i,j] = A[i]*B[j]
```

If A and B are higher-dimensional, the result has dimension A.ndim+B.ndim

Parameters

A : array_like

First term

B : array_like

Second term

Returns

r : ndarray

Output array

Examples

```
>>> np.multiply.outer([1,2,3],[4,5,6])
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])
```

Warning: A reduce-like operation on an array with a data type that has range “too small “to handle the result will silently wrap. You should use dtype to increase 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`). Nonetheless, 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 in 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

`add`(x1,x2[,out])	Add arguments element-wise.
`subtract`(x1,x2[,out])	Subtract arguments element-wise.
`multiply`(x1,x2[,out])	Multiply arguments elementwise.
`divide`(x1,x2[,out])	Divide arguments element-wise.
`logaddexp`(x1,x2[,out])	Logarithm of *exp(x)* + *exp(y)*.
`logaddexp2`(x1,x2[,out])	Base-2 Logarithm of *2**x* + *2**y*.
`true_divide`(x1,x2[,out])	Returns an element-wise, true division of the inputs.
`floor_divide`(x1,x2[,out])	Return the largest integer smaller or equal to the division of the inputs.
`negative`(x[,out])	Returns an array with the negative of each element of the original array.
`power`(x1,x2[,out])	Returns element-wise base array raised to power from second array.
`remainder`(x1,x2[,out])	Returns element-wise remainder of division.
`mod`(x1,x2[,out])	Returns element-wise remainder of division.
`fmod`(x1,x2[,out])	Return the remainder of division.
`absolute`(x[,out])	Calculate the absolute value element-wise.
`rint`(x[,out])	Round elements of the array to the nearest integer.
`sign`(x[,out])	Returns an element-wise indication of the sign of a number.
`conj`(x[,out])	Return the complex conjugate, element-wise.
`exp`(x[,out])	Calculate the exponential of the elements in the input array.
`exp2`(x[,out])	Calculate *2**p* for all *p* in the input array.
`log`(x[,out])	Natural logarithm, element-wise.
`log2`(x[,y])	Return the base 2 logarithm.
`log10`(x[,out])	Compute the logarithm in base 10 element-wise.
`expm1`(x[,out])	Return the exponential of the elements in the array minus one.
`log1p`(x[,out])	*log(1 + x)* in base *e*, elementwise.
`sqrt`(x[,out])	Return the positive square-root of an array, element-wise.
`square`(x[,out])	Return the element-wise square of the input.
`reciprocal`(x[,out])	Return element-wise reciprocal.
`ones_like`(x[,out])	Returns an array of ones with the same shape and type as a given array.

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 $180°/\pi$.

`sin`(x[,out])	Trigonometric sine, element-wise.
`cos`(x[,out])	Cosine elementwise.
`tan`(x[,out])	Compute tangent element-wise.
`arcsin`(x[,out])	Inverse sine elementwise.
`arccos`(x[,out])	Trigonometric inverse cosine, element-wise.
`arctan`(x[,out])	Trigonometric inverse tangent, element-wise.
`arctan2`(x1,x2[,out])	Elementwise arc tangent of `x1/x2` choosing the quadrant correctly.
`hypot`(x1,x2[,out])	Given two sides of a right triangle, return its hypotenuse.
`sinh`(x[,out])	Hyperbolic sine, element-wise.
`cosh`(x[,out])	Hyperbolic cosine, element-wise.
`tanh`(x[,out])	Hyperbolic tangent element-wise.
`arcsinh`(x[,out])	Inverse hyperbolic sine elementwise.
`arccosh`(x[,out])	Inverse hyperbolic cosine, elementwise.
`arctanh`(x[,out])	Inverse hyperbolic tangent elementwise.
`deg2rad`(x[,out])	Convert angles from degrees to radians. This is the same function as radians, but deg2rad is a more descriptive name.
`rad2deg`(x[,out])	Convert angles from radians to degrees. This is the same function as degrees but is preferred because its more descriptive name.

2.7.3 Bit-twiddling functions

These function all need integer arguments and they maniuplate the bit- pattern of those arguments.

`bitwise_and`(x1,x2[,out])	Compute bit-wise AND of two arrays, element-wise.
`bitwise_or`(x1,x2[,out])	Compute bit-wise OR of two arrays, element-wise.
`bitwise_xor`(x1,x2[,out])	Compute bit-wise XOR of two arrays, element-wise.
`invert`(x[,out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
`left_shift`(x1,x2[,out])	Shift the bits of an integer to the left.
`right_shift`(x1,x2[,out])	Shift the bits of an integer to the right.

2.7.4 Comparison functions

`greater`(x1,x2[,out])	Return (x1 > x2) element-wise.
`greater_equal`(x1,x2[,out])	Element-wise True if first array is greater or equal than second array.
`less`(x1,x2[,out])	Returns (x1 < x2) element-wise.
`less_equal`(x1,x2[,out])	Returns (x1 <= x2) element-wise.
`not_equal`(x1,x2[,out])	Return (x1 != x2) element-wise.
`equal`(x1,x2[,out])	Returns elementwise x1 == x2 in a bool array.

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.

`logical_and`(x1,x2[,out])	Compute the truth value of x1 AND x2 elementwise.
`logical_or`(x1,x2[,out])	Compute the truth value of x1 OR x2 elementwise.
`logical_xor`(x1,x2[,out])	Compute the truth value of x1 XOR x2 elementwise.
`logical_not`(x[,out])	Compute the truth value of NOT x elementwise.

Warning: The Bitwise operators (& and |) are the proper way to combine element-by-element array comparisons. Be sure to 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.

`maximum`(x1,x2[,out])	Element-wise maximum of array elements.

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 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.

isreal(x)	Returns a bool array where True if the corresponding input element is real.
iscomplex(x)	Return a bool array, True if element is complex (non-zero imaginary part).
isfinite(x[,out])	Returns True for each element that is a finite number.
isinf(x[,out])	Shows which elements of the input are positive or negative infinity. Returns a numpy boolean scalar or array resulting from an element-wise test for positive or negative infinity.
isnan(x[,out])	Returns a numpy boolean scalar or array resulting from an element-wise test for Not a Number (NaN).
signbit(x[,out])	Returns element-wise True where signbit is set (less than zero).
modf(x[,out1,out2])	Return the fractional and integral part of a number.
ldexp(x1,x2[,out])	Compute y = x1 * 2**x2.
frexp(x[,out1,out2])	Split the number, x, into a normalized fraction (y1) and exponent (y2)
fmod(x1,x2[,out])	Return the remainder of division.
floor(x[,out])	Return the floor of the input, element-wise.
ceil(x[,out])	Return the ceiling of the input, element-wise.
trunc(x[,out])	Return the truncated value of the input, element-wise.

CHAPTER

THREE

ROUTINES

3.1 Array creation routines

See Also:

Array creation (in *NumPy User Guide*)

3.1.1 Ones and zeros

`empty`(shape[,dtype,order])	Return a new array of given shape and type, without initialising entries.
`empty_like`(a)	Create a new array with the same shape and type as another.
`eye`(N[,M,k,dtype])	Return a 2-D array with ones on the diagonal and zeros elsewhere.
`identity`(n[,dtype])	Return the identity array.
`ones`(shape[,dtype,order])	Return a new array of given shape and type, filled with ones.
`ones_like`(x[,out])	Returns an array of ones with the same shape and type as a given array.
`zeros`(shape[,dtype,order])	Return a new array of given shape and type, filled with zeros.
`zeros_like`(a)	Returns an array of zeros with the same shape and type as a given array.

`empty` (*shape, dtype=float, order='C'*)

Return a new array of given shape and type, without initialising entries.

Parameters

shape : {tuple of int, 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`

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],  #random data
       [  2.13182611e-314,   3.06959433e-309]])
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],  #random data
       [  496041986,    19249760]])
```

empty_like(*a*)
Create a new array with the same shape and type as another.

Parameters
a : ndarray
Returned array will have same shape and type as *a*.

See Also:

`zeros_like`, `ones_like`, `zeros`, `ones`, `empty`

Notes

This function does *not* initialize the returned array; to do that use *zeros_like* or *ones_like* instead.

Examples

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.empty_like(a)
>>> np.empty_like(a)
array([[-1073741821, -1067702173,       65538],    #random data
       [      25670,    23454291,       71800]])
```

eye(*N, M=None, k=0, dtype=<type 'float'>*)
Return a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters
N : int
Number of rows in the output.
M : int, optional
Number of columns in the output. If None, defaults to *N*.
k : int, optional
Index of the diagonal: 0 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 (*x, [out]*)

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 (*shape, dtype=float, order='C'*)

Return a new array of given shape and type, filled with zeros.

Parameters

shape : {tuple of ints, int}

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:

`numpy.zeros_like`
Return an array of zeros with shape and type of input.

`numpy.ones_like`
Return an array of ones with shape and type of input.

`numpy.empty_like`
Return an empty array with shape and type of input.

`numpy.ones`
Return a new array setting values to one.

`numpy.empty`
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')])
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

zeros_like(*a*)

Returns 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* defines the parameters of the returned array.

Returns

out : ndarray

Array of zeros with same shape and type as *a*.

See Also:

numpy.ones_like
Return an array of ones with shape and type of input.

numpy.empty_like
Return an empty array with shape and type of input.

numpy.zeros
Return a new array setting values to zero.

numpy.ones
Return a new array setting values to one.

numpy.empty
Return a new uninitialized array.

Examples

```
>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.zeros_like(x)
array([[0, 0, 0],
       [0, 0, 0]])
```

3.1.2 From existing data

`array`(object[,dtype,copy,order,...])	Create an array.
`asarray`(a[,dtype,order])	Convert the input to an array.
`asanyarray`(a[,dtype,order])	Convert the input to a ndarray, but pass ndarray subclasses through.
`ascontiguousarray`(a[,dtype])	Return a contiguous array in memory (C order).
`asmatrix`(data[,dtype])	Interpret the input as a matrix.
`copy`(a)	Return an array copy of the given object.
`frombuffer`(buffer[,dtype,count,offset])	Interpret a buffer as a 1-dimensional array.
`fromfile`(file[,dtype,count,sep])	Construct an array from data in a text or binary file.
`fromfunction`(function,shape,**kwargs)	Construct an array by executing a function over each coordinate.
`fromiter`(iterable,dtype[,count])	Create a new 1-dimensional array from an iterable object.
`loadtxt`(fname[,dtype,comments,...])	Load data from a text file.

`array` (*object, dtype=None, copy=True, order=None, subok=True, ndmin=True*)
Create an array.

Parameters

object : array_like

An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype : data-type, optional

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy : bool, optional

If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*dtype*, *order*, etc.).

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

Specify the order of the array. If order is 'C' (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.

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:

```
>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Data-type consisting of more than one element:

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
```

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 ('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:

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

Existing arrays are not copied:

```
>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True
```

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:

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

Instances of *ndarray* subclasses are passed through as-is:

```
>>> a = np.matrix([1, 2])
>>> np.asanyarray(a) is a
True
```

ascontiguousarray(*a, dtype=None*)
Return a contiguous array in memory (C order).

Parameters
a : array_like
Input array.
dtype : string
Type code of returned array.
Returns
out : ndarray
Contiguous array of same shape and content as *a* with type *dtype*.

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:

```
>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)
```

Note that, when we modify x, y changes, but not z:

```
>>> x[0] = 10
>>> x[0] == y[0]
True
>>> x[0] == z[0]
False
```

frombuffer (*buffer, dtype=float, count=-1, offset=0*)

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.:

```
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

fromfile (*file, dtype=float, count=-1, sep=''*)

Construct an array from data in a text or binary file.

A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the *tofile* method can be read using this function.

Parameters

file : file or string

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 : string

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:

```
>>> dt = np.dtype([('time', [('min', int), ('sec', int)]),
...                ('temp', float)])
>>> x = np.zeros((1,), dtype=dt)
>>> x['time']['min'] = 10; x['temp'] = 98.25
>>> x
array([((10, 0), 98.25)],
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])
```

Save the raw data to disk:

```
>>> import os
>>> fname = os.tmpnam()
>>> x.tofile(fname)
```

Read the raw data from disk:

```
>>> np.fromfile(fname, dtype=dt)
array([((10, 0), 98.25)],
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])
```

The recommended way to store and load data:

```
>>> np.save(fname, x)
>>> np.load(fname + '.npy')
array([((10, 0), 98.25)],
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])
```

fromfunction (*function, shape, **kwargs*)

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value `fn(x, y, z)` at coordinate `(x, y, z)`.

Parameters

fn : 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]]`. *fn* 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 *fn*.

dtype : data-type, optional

Data-type of the coordinate arrays passed to *fn*. By default, *dtype* is float.

See Also:

`indices`, `meshgrid`

Notes

Keywords other than *shape* and *dtype* are passed to the 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 (*iterable, dtype, count=-1*)

Create a new 1-dimensional array from an iterable object.

Parameters

iterable : iterable object

An iterable object providing data for the array.

dtype : data-type

The data type of the returned array.

count : int, optional

The number of items to read from iterable. The default is -1, which means all data is read.

Returns

out : ndarray

The output array.

Notes

Specify `count` to improve performance. It allows `fromiter` to pre-allocate the output array, instead of resizing it on demand.

Examples

```
>>> iterable = (x*x for x in range(5))
>>> np.fromiter(iterable, np.float)
array([  0.,   1.,   4.,   9.,  16.])
```

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 string

File or filename to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.

dtype : data-type

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 : string, optional

The character used to indicate the start of a comment.

delimiter : string, optional

The string used to separate values. By default, this is any whitespace.

converters : {}

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

Skip the first *skiprows* lines.

usecols : sequence

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

If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`

Returns

out : ndarray

Data read from the text file.

See Also:

scipy.io.loadmat
reads Matlab(R) data files

Examples

```
>>> 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={'names': ('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.])
```

3.1.3 Creating record arrays (numpy.rec)

Note: numpy.rec is the preferred alias for numpy.core.records.

core.records.array(obj[,dtype,shape,offset,...])	Construct a record array from a wide-variety of objects.
core.records.fromarrays(arrayList[,dtype,shape,formats,...])	create a record array from a (flat) list of arrays
core.records.fromrecords(recList[,dtype,shape,formats,...])	create a recarray from a list of records in text form
core.records.fromstring(datastring[,dtype,shape,offset,...])	create a (read-only) record array from binary data contained in a string
core.records.fromfile(fd[,dtype,shape,offset,...])	Create an array from binary file data

array (*obj, dtype=None, shape=None, offset=0, strides=None, formats=None, names=None, titles=None, aligned=False, byteorder=None, copy=True*)
Construct a record array from a wide-variety of objects.

fromarrays (*arrayList, dtype=None, shape=None, formats=None, names=None, titles=None, aligned=False, byteorder=None*)
create a record array from a (flat) list of arrays

```
>>> x1=np.array([1,2,3,4])
>>> x2=np.array(['a','dd','xyz','12'])
>>> x3=np.array([1.1,2,3,4])
>>> r = np.core.records.fromarrays([x1,x2,x3],names='a,b,c')
>>> print r[1]
(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('<')
>>> 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
(10,)
```

3.1.4 Creating character arrays (`numpy.char`)

Note: `numpy.char` is the preferred alias for `numpy.core.defchararray`.

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

`array` (*obj, itemsize=None, copy=True, unicode=False, order=None*)

3.1.5 Numerical ranges

`arange`([start,]stop[,step,...])	Return evenly spaced values within a given interval.
`linspace`(start,stop[,num,endpoint,retstep])	Return evenly spaced numbers over a specified interval.
`logspace`(start,stop[,num,endpoint,base])	Return numbers spaced evenly on a log scale.
`meshgrid`(x,y)	Return coordinate matrices from two coordinate vectors.
`mgrid`	Construct a multi-dimensional filled "meshgrid".

`arange` (*[start], stop, [step], dtype=None*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in range 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

```
>>> 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.

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
: Similiar 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
Similiar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.

linspace
Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.

Notes

Logspace is equivalent to the code

```
>>> y = linspace(start, stop, num=num, endpoint=endpoint)
>>> power(base, y)
```

Examples

```
>>> np.logspace(2.0, 3.0, num=4)
    array([  100.        ,   215.443469  ,   464.15888336,  1000.        ])
>>> np.logspace(2.0, 3.0, num=4, endpoint=False)
    array([ 100.        ,  177.827941  ,  316.22776602,  562.34132519])
>>> np.logspace(2.0, 3.0, num=4, base=2.0)
    array([ 4.        ,  5.0396842 ,  6.34960421,  8.        ])
```

Graphical illustration:

```
>>> import matplotlib.pyplot as plt
>>> N = 10
>>> x1 = np.logspace(0.1, 1, N, endpoint=True)
>>> x2 = np.logspace(0.1, 1, N, endpoint=False)
>>> y = np.zeros(N)
>>> plt.plot(x1, y, 'o')
>>> plt.plot(x2, y + 0.5, 'o')
>>> plt.ylim([-0.5, 1])
>>> plt.show()
```

meshgrid(*x, y*)

Return coordinate matrices from two coordinate vectors.

Parameters

x, y : ndarray

Two 1D arrays representing the x and y coordinates

Returns

X, Y : ndarray

For vectors *x*, *y* with lengths Nx=len(*x*) and Ny=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:

numpy.mgrid
Construct a multi-dimensional "meshgrid" using indexing notation.

Examples

```
>>> X, Y = numpy.meshgrid([1,2,3], [4,5,6,7])
>>> X
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
>>> Y
array([[4, 4, 4],
       [5, 5, 5],
       [6, 6, 6],
       [7, 7, 7]])
```

mgrid()

Construct a multi-dimensional filled "meshgrid".

Returns a mesh-grid when indexed. 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**.

See Also:

`ogrid`

Examples

```
>>> 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]],
<BLANKLINE>
       [[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. ])
```

3.1.6 Building matrices

`diag`(v[,k])	Extract a diagonal or construct a diagonal array.
`diagflat`(v[,k])	Create a 2-dimensional array with the flattened input as a diagonal.
`tri`(N[,M,k,dtype])	Construct an array filled with ones at and below the given diagonal.
`tril`(m[,k])	Lower triangle of an array.
`triu`(m[,k])	Upper triangle of an array.
`vander`(x[,N])	Generate a Van der Monde matrix.

diag(*v, k=0*)

Extract a diagonal or construct a diagonal array.

Parameters

v : array_like

If *v* is a 2-dimensional array, return a copy of its *k*-th diagonal. If *v* is a 1-dimensional array, return a 2-dimensional array with *v* on the *k*-th diagonal.

k : int, optional

Diagonal in question. The defaults is 0.

Examples

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> np.diag(x)
array([0, 4, 8])
```

```
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

`diagflat`(*v, k=0*)

Create a 2-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.

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

```
>>> np.tri(3, 5, 2, dtype=int)
array([[1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1]])
```

```
>>> np.tri(3, 5, -1)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.]])
```

tril(*m, k=0*)

Lower triangle of an array.

Return a copy of an array with elements above the *k*-th diagonal zeroed.

Parameters

m : array_like, shape (M, N)

Input array.

k : int

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

```
>>> 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.

Please refer to the documentation for *tril*.

See Also:

`tril`

Examples

```
>>> np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 0,  8,  9],
       [ 0,  0, 12]])
```

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`.

Parameters

x : array_like

Input array.

N : int, optional

Order of (number of columns in) the output.

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.

Examples

```
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

3.1.7 The Matrix class

`mat`(data[,dtype])	Interpret the input as a matrix.
`bmat`(obj[,ldict,gdict])	Build a matrix object from a string, nested sequence, or array.

`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

```
>>> 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

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

All the following expressions construct the same block matrix:

```
>>> np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
>>> np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
>>> np.bmat('A,B; C,D')
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```

3.2 Array manipulation routines

3.2.1 Changing array shape

`reshape`(a,newshape[,order])	Gives a new shape to an array without changing its data.
`ravel`(a[,order])	Return a flattened array.
`ndarray.flat`	A 1-d flat iterator.
`ndarray.flatten`([order])	Collapse an array into one dimension.

`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 : {tuple, int}

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

```
>>> 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-dimensional 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(),)` (or `(np.prod(a.shape),)`).

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 generalised 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)`:

```
>>> 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

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

`flat`

A 1-d flat iterator.

Examples

```
>>> x = np.arange(3*4*5)
>>> x.shape = (3,4,5)
>>> x.flat[19]
19
>>> x.T.flat[19]
31
```

`flatten`(*order='C'*)

Collapse an array 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.

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

`rollaxis`(a,axis[,start])	Roll the specified axis backwards, until it lies in a given position.
`swapaxes`(a,axis1,axis2)	Interchange two axes of an array.
`ndarray.T`	Same as self.transpose() except self is returned for self.ndim < 2.
`transpose`(a[,axes])	Permute the dimensions of an array.

`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.

Examples

```
>>> a = np.ones((3,4,5,6))
>>> np.rollaxis(a, 3, 1).shape
(3, 6, 4, 5)
>>> np.rollaxis(a, 2).shape
(5, 3, 4, 6)
>>> np.rollaxis(a, 1, 4).shape
(3, 5, 6, 4)
```

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

```
>>> 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]],
<BLANKLINE>
       [[4, 5],
        [6, 7]]])
```

```
>>> np.swapaxes(x,0,2)
array([[[0, 4],
        [2, 6]],
<BLANKLINE>
       [[1, 5],
        [3, 7]]])
```

T

Same as self.transpose() except self is returned for self.ndim < 2.

Examples

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

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])
```

```
>>> np.transpose(x)
array([[0, 2],
       [1, 3]])
```

```
>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)
```

3.2.3 Changing number of dimensions

`atleast_1d`(*arys)	Convert inputs to arrays with at least one dimension.
`atleast_2d`(*arys)	View inputs as arrays with at least two dimensions.
`atleast_3d`(*arys)	View inputs as arrays with at least three dimensions.
`broadcast`	
`broadcast_arrays`(*args)	Broadcast any number of arrays against each other.
`expand_dims`(a,axis)	Expand the shape of an array.
`squeeze`(a)	Remove single-dimensional entries from the shape of an array.

`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

```
>>> 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

```
>>> numpy.atleast_2d(3.0)
array([[ 3.]])
```

```
>>> x = numpy.arange(3.0)
>>> numpy.atleast_2d(x)
array([[ 0.,  1.,  2.]])
>>> numpy.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 one-dimensional array of shape `N` becomes a view of shape `(1, N, 1)`. An `(M, N)` array becomes a view of shape `(N, M, 1)`.

See Also:

`numpy.atleast_1d`, `numpy.atleast_2d`

Examples

```
>>> numpy.atleast_3d(3.0)
array([[[ 3.]]])
```

```
>>> x = numpy.arange(3.0)
>>> numpy.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = numpy.arange(12.0).reshape(4,3)
>>> numpy.atleast_3d(x).shape
(4, 3, 1)
>>> numpy.atleast_3d(x).base is x
True
```

```
>>> for arr in np.atleast_3d(1, [1, 2], [[1, 2]]): print arr, "\n"
...
[[[1]]]
```

[[[1]

[2]]]

[[[1]

[2]]]

class **broadcast**()

broadcast_arrays(**args*)

Broadcast any number of arrays against each other.

Parameters

'*args' : arrays

The arrays to broadcast.

Returns

broadcasted : list of arrays

These arrays are views on the original arrays. They are typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location. If you need to write to the arrays, make copies first.

Examples

```
>>> x = np.array([[1,2,3]])
>>> y = np.array([[1],[2],[3]])
>>> np.broadcast_arrays(x, y)
[array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]), array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])]
```

Here is a useful idiom for getting contiguous copies instead of non-contiguous views.

```
>>> 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

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

The following is equivalent to `x[np.newaxis,:]` or `x[np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=0)
>>> y
array([[1, 2]])
>>> y.shape
(1, 2)
```

```
>>> y = np.expand_dims(x, axis=1)  # Equivalent to x[:,newaxis]
>>> y
array([[1],
       [2]])
>>> y.shape
(2, 1)
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects:

```
>>> np.newaxis is None
True
```

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

```
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
```

3.2.4 Changing kind of array

asarray(a[,dtype,order])	Convert the input to an array.
asanyarray(a[,dtype,order])	Convert the input to a ndarray, but pass ndarray subclasses through.
asmatrix(data[,dtype])	Interpret the input as a matrix.
asfarray(a[,dtype])	Return an array converted to float type.
asfortranarray(a[,dtype])	Return an array laid out in Fortran-order in memory.
asscalar(a)	Convert an array of size 1 to its scalar equivalent.
require(a[,dtype,requirements])	Return an ndarray of the provided type that satisfies requirements.

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 ('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:

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

Existing arrays are not copied:

```
>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True
```

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:

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

Instances of *ndarray* subclasses are passed through as-is:

```
>>> a = np.matrix([1, 2])
>>> np.asanyarray(a) is a
True
```

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]])
```

asfarray(*a, dtype=<type 'numpy.float64'>*)
Return an array converted to float type.

Parameters
a : array_like

Input array.

dtype : string or dtype object, optional

Float type code to coerce input array *a*. If one of the 'int' dtype, it is replaced with float64.

Returns

out : ndarray, float

Input *a* as a float ndarray.

Examples

```
>>> np.asfarray([2, 3])
array([ 2.,  3.])
>>> np.asfarray([2, 3], dtype='float')
array([ 2.,  3.])
>>> np.asfarray([2, 3], dtype='int8')
array([ 2.,  3.])
```

asfortranarray(*a, dtype=None*)

Return an array laid out in Fortran-order in memory.

Parameters

a : array_like

Input array.

dtype : data-type, optional

By default, the data-type is inferred from the input data.

Returns

out : ndarray

Array interpretation of *a* in Fortran (column-order).

See Also:

asarray
Similar function which always returns ndarrays.

ascontiguousarray
Convert input to a contiguous array.

asfarray
Convert input to a floating point ndarray.

asanyarray
Convert input to an ndarray with either row or 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.

asscalar(*a*)

Convert an array of size 1 to its scalar equivalent.

Parameters

a : ndarray

Input array.

Returns

out : scalar

Scalar of size 1 array.

Examples

```
>>> 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 (None is the default data-type – float64)

requirements : list of strings

The requirements list can be any of the following

- 'ENSUREARRAY' ('E') - ensure that a base-class ndarray
- '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 writeable array
- 'OWNDATA' ('O') - ensure an array that owns its own data

Notes

The returned array will be guaranteed to have the listed requirements by making a copy if needed.

3.2.5 Joining arrays

`append`(arr,values[,axis])	Append values to the end of an array.
`column_stack`(tup)	Stack 1-D arrays as columns into a 2-D array
`concatenate`((a1,a2,...)[,axis])	Join a sequence of arrays together.
`dstack`(tup)	Stack arrays in sequence depth wise (along third axis)
`hstack`(tup)	Stack arrays in sequence horizontally (column wise)
`vstack`(tup)	Stack arrays vertically.

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.

Examples

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

column_stack (*tup*)

Stack 1-D arrays as columns into a 2-D array

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters

tup : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

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 (*(a1, a2, ...), axis=0*)

Join a sequence of arrays together.

Parameters

a1, a2, ... : sequence of ndarrays

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:

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)

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

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

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
<BLANKLINE>
       [[2, 3]],
<BLANKLINE>
       [[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 along first axis.

dstack
Stack along third axis.

concatenate
Join arrays.

hsplit
Split array along second axis.

Notes

Equivalent to `np.concatenate(tup, axis=1)`

Examples

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

vstack(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
: Split an array into a list of multiple sub-arrays of near-equal size.

split
: Split array into a list of multiple sub-arrays of equal size.

vsplit
: Split array into a list of multiple sub-arrays vertically.

dsplit
: Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
: Join arrays together.

hstack
: Stack arrays in sequence horizontally (column wise).

dstack
: Stack arrays in sequence depth wise (along third dimension).

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]])
```

3.2.6 Splitting arrays

`array_split`(ary,indices_or_sections[,axis])	Split an array into multiple sub-arrays of equal or near-equal size.
`dsplit`(ary,indices_or_sections)	Split array into multiple sub-arrays along the 3rd axis (depth).
`hsplit`(ary,indices_or_sections)	Split array into multiple sub-arrays horizontally.
`split`(ary,indices_or_sections[,axis])	Split an array into multiple sub-arrays of equal size.
`vsplit`(ary,indices_or_sections)	Split array into multiple sub-arrays vertically.

`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 `numpy.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:

`numpy.split`
Split array into multiple sub-arrays.

Examples

```
>>> 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).

Parameters

ary : ndarray
An array, with at least 3 dimensions, to be divided into sub-arrays depth-wise, or along the third axis.

indices_or_sections: integer or 1D array :
If *indices_or_sections* is an integer, N, the array will be divided into N equal arrays along *axis*. If an equal split is not possible, a ValueError is raised.
if *indices_or_sections* is a 1D array of sorted integers representing indices along *axis*, the array will be divided such that each index marks the start of each sub-array. If an index exceeds the dimension of the array along *axis*, and empty sub-array is returned for that index.

axis : integer, optional
the axis along which to split. Default is 0.

Returns

sub-arrays : list
A list of sub-arrays.

See Also:

`array_split`
Split an array into a list of multiple sub-arrays of near-equal size.

`split`
Split array into a list of multiple sub-arrays of equal size.

hsplit
: Split array into a list of multiple sub-arrays horizontally

vsplit
: Split array into a list of multiple sub-arrays vertically

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)

Notes

dsplit requires that sub-arrays are of equal shape, whereas *array_split* allows for sub-arrays to have nearly-equal shape. Equivalent to *split* with *axis* = 2.

Examples

```
>>> x = np.arange(16.0).reshape(2, 2, 4)
>>> np.dsplit(x, 2)
<BLANKLINE>
[array([[[  0.,   1.],
        [  4.,   5.]],
<BLANKLINE>
       [[  8.,   9.],
        [ 12.,  13.]]]),
 array([[[  2.,   3.],
        [  6.,   7.]],
<BLANKLINE>
       [[ 10.,  11.],
        [ 14.,  15.]]])]
<BLANKLINE>
>>> x = np.arange(16.0).reshape(2, 2, 4)
>>> np.dsplit(x, array([3, 6]))
<BLANKLINE>
[array([[[  0.,   1.,   2.],
        [  4.,   5.,   6.]],
<BLANKLINE>
       [[  8.,   9.,  10.],
        [ 12.,  13.,  14.]]]),
 array([[[  3.],
        [  7.]],
<BLANKLINE>
       [[ 11.],
        [ 15.]]]),
 array([], dtype=float64)]
```

hsplit(*ary, indices_or_sections*)

Split array into multiple sub-arrays horizontally.

Please refer to the `numpy.split` documentation. *hsplit* is equivalent to `numpy.split` with `axis = 1`.

See Also:

split
: Split array into multiple sub-arrays.

Examples

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> np.hsplit(x, 2)
<BLANKLINE>
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [ 12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [ 10.,  11.],
       [ 14.,  15.]])]
```

```
>>> np.hsplit(x, array([3, 6]))
<BLANKLINE>
[array([[  0.,   1.,   2.],
       [  4.,   5.,   6.],
       [  8.,   9.,  10.],
       [ 12.,  13.,  14.]]),
 array([[  3.],
       [  7.],
       [ 11.],
       [ 15.]]),
 array([], dtype=float64)]
```

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: integer or 1D 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 1D 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 : integer, optional

The axis along which to split. Default is 0.

Returns

sub-arrays : list

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

```
>>> 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])
<BLANKLINE>
[array([ 0.,  1.,  2.]),
 array([ 3.,  4.]),
 array([ 5.]),
 array([ 6.,  7.]),
 array([], dtype=float64)]
```

vsplit (*ary, indices_or_sections*)
Split array into multiple sub-arrays vertically.

Please refer to the `numpy.split` documentation.

See Also:

numpy.split
The default behaviour of this function implements *vsplit*.

3.2.7 Tiling arrays

`tile`(A,reps)	Construct an array by repeating A the number of times given by reps.
`repeat`(a,repeats[,axis])	Repeat elements of an array.

tile (*A, reps*)
Construct an array by repeating A the number of times given by reps.

Parameters

A : array_like

The input array.

reps : array_like

The number of repetitions of *A* along each axis.

Returns

c : ndarray

The output array.

See Also:

`repeat`

Notes

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).

Examples

```
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> np.tile(a, (2, 1, 2))
array([[[0, 1, 2, 0, 1, 2]],
<BLANKLINE>
       [[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, 1, 2, 2, 3, 3, 4, 4])
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 1, 2, 2, 2],
       [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

`delete`(arr,obj[,axis])	Return a new array with sub-arrays along an axis deleted.
`insert`(arr,obj,values[,axis])	Insert values along the given axis before the given indices.
`resize`(a,new_shape)	Return a new array with the specified shape.
`trim_zeros`(filt[,trim])	Trim the leading and trailing zeros from a 1D array.
`unique`(x)	Return the sorted, unique elements of an array or sequence.

delete (*arr, obj, axis=None*)

Return a new array with sub-arrays along an axis deleted.

Parameters

arr : array_like

Input array.

obj : slice, integer or an array of integers

Indicate which sub-arrays to remove.

axis : integer, optional

The axis along which to delete the subarray defined by *obj*. If *axis* is None, *obj* is applied to the flattened array.

See Also:

insert
Insert values into an array.

append
Append values at the end of an array.

Examples

```
>>> arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.delete(arr, 1, 0)
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
>>> np.delete(arr, np.s_[::2], 1)
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>> np.delete(arr, [1,3,5], None)
array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
```

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

Insert values along the given axis before the given indices.

Parameters

arr : array_like

Input array.

obj : {integer, slice, integer array_like}

Insert *values* before *obj* indices.

values : :

Values to insert into *arr*.

axis : int, optional

Axis along which to insert *values*. If *axis* is None then ravel *arr* first.

Examples

```
>>> a = np.array([[1,2,3],
...               [4,5,6],
...               [7,8,9]])
>>> np.insert(a, [1,2], [[4],[5]], axis=0)
array([[1, 2, 3],
       [4, 4, 4],
       [4, 5, 6],
       [5, 5, 5],
       [7, 8, 9]])
```

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.

See Also:

`ndarray.resize`
resize an array in-place.

Examples

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

`trim_zeros` (*filt, trim='fb'*)
Trim the leading and trailing zeros from a 1D array.

Parameters
filt : array_like
Input array.
trim : string, optional
A string with 'f' representing trim from front and 'b' to trim from back.

Examples

```
>>> a = np.array((0, 0, 0, 1, 2, 3, 2, 1, 0))
>>> np.trim_zeros(a)
array([1, 2, 3, 2, 1])
```

`unique` (*x*)
Return the sorted, unique elements of an array or sequence.

Parameters
x : array_like
Input array.

Returns
y : ndarray
The sorted, unique elements are returned in a 1-D array.

Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

3.2.9 Rearranging elements

`fliplr`(m)	Left-right flip.
`flipud`(m)	Up-down flip.
`reshape`(a,newshape[,order])	Gives a new shape to an array without changing its data.
`roll`(a,shift[,axis])	Roll array elements along a given axis.
`rot90`(m[,k])	Rotate an array by 90 degrees in the counter-clockwise direction.

fliplr(*m*)

Left-right flip.

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 $\mathcal{O}(1)$.

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

```
>>> A = np.diag([1.,2.,3.])
>>> A
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*)

Up-down flip.

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 $\mathcal{O}(1)$.

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 : {tuple, int}

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

```
>>> 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

```
>>> 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

```
>>> m = np.array([[1,2],[3,4]], int)
>>> m
array([[1, 2],
       [3, 4]])
>>> np.rot90(m)
array([[2, 4],
       [1, 3]])
>>> np.rot90(m, 2)
array([[4, 3],
       [2, 1]])
```

3.3 Indexing routines

See Also:

Indexing

3.3.1 Generating index arrays

`c_`	Translates slice objects to concatenation along the second axis.
`r_`	Translates slice objects to concatenation along the first axis.
`s_`	A nicer way to build up index tuples for arrays.
`nonzero`(a)	Return the indices of the elements that are non-zero.
`where`(condition,[x,y])	Return elements, either from *x* or *y*, depending on *condition*.
`indices`(dimensions[,dtype])	Return an array representing the indices of a grid.
`ix_`(*args)	Construct an open mesh from multiple sequences.
`ogrid`	Construct a multi-dimensional open “meshgrid”.
`unravel_index`(x,dims)	Convert a flat index into an index tuple for an array of given shape.

c_()

Translates slice objects to concatenation along the second axis.

For example: >>> np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])] array([1, 2, 3, 0, 0, 4, 5, 6])

r_()

Translates slice objects to concatenation along the first axis.

For example: >>> np.r_[np.array([1,2,3]), 0, 0, np.array([4,5,6])] array([1, 2, 3, 0, 0, 4, 5, 6])

s_()

A nicer way to build up index tuples for arrays.

For any index combination, including slicing and axis insertion, ‘a[indices]’ is the same as ‘a[index_exp[indices]]’ for any array ‘a’. However, ‘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.

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

```
>>> x = np.eye(3)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))
```

```
>>> x[np.nonzero(x)]
array([ 1.,  1.,  1.])
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])
```

where (*condition, [x, y]*)
Return elements, either from *x* or *y*, depending on *condition*.

If only *condition* is given, return `condition.nonzero()`.

Parameters
condition : array_like, bool
When True, yield *x*, otherwise yield *y*.
x, y : array_like, optional
Values from which to choose.

Returns
out : ndarray or tuple of ndarrays
If both *x* and *y* are specified, the output array, shaped like *condition*, 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

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

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

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 : 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]])
```

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[ix_([1,3,7],[2,5,8])] returns the array

a[1,2] a[1,5] a[1,8] a[3,2] a[3,5] a[3,8] a[7,2] a[7,5] a[7,8]

ogrid()

Construct a multi-dimensional open "meshgrid".

Returns an 'open' mesh-grid when indexed. 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.

The returned mesh-grid is open (or not fleshed out), so that only one-dimension of each returned argument is greater than 1

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**.

See Also:

`mgrid`

Examples

```
>>> np.ogrid[0:5,0:5]
[array([[0],
        [1],
        [2],
        [3],
        [4]]), array([[0, 1, 2, 3, 4]])]
```

unravel_index(*x, dims*)

Convert a flat index into an index tuple for an array of given shape.

Parameters

x : int

Flattened index.

dims : shape tuple

Input shape.

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.ones((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
```

3.3.2 Indexing-like operations

take(a,indices[,axis,out,mode])	Take elements from an array along an axis.
choose(a,choices[,out,mode])	Use an index array to construct a new array from a set of choices.
compress(condition,a[,axis,out])	Return selected slices of an array along given axis.
diag(v[,k])	Extract a diagonal or construct a diagonal array.
diagonal(a[,offset,axis1,axis2])	Return specified diagonals.
select(condlist,choicelist[,default])	Return an array drawn from elements in choicelist, depending on conditions.

take (*a, indices, axis=None, out=None, mode='raise'*)

Take elements from an array along an axis.

This function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis.

Parameters

a : array_like

The source array.

indices : array_like, int

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 'wrap' – wrap around 'clip' – clip to the range

Returns

subarray : ndarray

The returned array has the same type as *a*.

See Also:

ndarray.take
equivalent method

Examples

```
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> np.take(a, indices)
array([4, 3, 6])
```

In this example if *a* is a ndarray, "fancy" indexing can be used. >>> a = np.array(a) >>> a[indices] array([4, 3, 6])

choose (*a, 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 function will create a new array that merges each of the choice arrays. Where a value in *a* is i, then the new array will have the value that choices[i] contains in the same place.

Parameters

a : int array

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

The merged results.

See Also:

ndarray.choose
equivalent method

Examples

```
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices)
array([20, 31, 12,  3])
>>> np.choose([2, 4, 1, 0], choices, mode='clip')
array([20, 31, 12,  3])
>>> np.choose([2, 4, 1, 0], choices, mode='wrap')
array([20,  1, 12,  3])
```

compress(*condition, a, axis=None, out=None*)

Return selected slices of an array along given axis.

Parameters

condition : array_like

Boolean 1-D array selecting 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:

`ndarray.compress`
Equivalent method.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.compress([0, 1], a, axis=0)
array([[3, 4]])
>>> np.compress([1], a, axis=1)
array([[1],
       [3]])
>>> np.compress([0,1,1], a)
array([2, 3])
```

diag (*v, k=0*)
Extract a diagonal or construct a diagonal array.

Parameters

v : array_like

If *v* is a 2-dimensional array, return a copy of its *k*-th diagonal. If *v* is a 1-dimensional array, return a 2-dimensional array with *v* on the *k*-th diagonal.

k : int, optional

Diagonal in question. The defaults is 0.

Examples

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> np.diag(x)
array([0, 4, 8])
```

```
>>> np.diag(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

```
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])
```

```
>>> 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 N boolean arrays of length M

The conditions C_0 through C_(N-1) which determine from which vector the output elements are taken.

choicelist : list of N arrays of length M

Th vectors V_0 through V_(N-1), from which the output elements are chosen.

Returns

output : 1-dimensional array of length M

The output at position m is the m-th element of the first vector V_n for which C_n[m] is non-zero. Note that the output depends on the order of conditions, since the first satisfied condition is used.

Notes

Equivalent to:

```
output = []
for m in range(M):
    output += [V[m] for V,C in zip(values,cond) if C[m]]
              or [default]
```

Examples

```
>>> t = np.arange(10)
>>> s = np.arange(10)*100
>>> condlist = [t == 4, t > 5]
>>> choicelist = [s, t]
>>> np.select(condlist, choicelist)
array([  0,    0,    0,    0, 400,    0,    6,    7,    8,    9])
```

3.3.3 Inserting data into arrays

place(arr,mask,vals)	Changes elements of an array based on conditional and input values.
put(a,ind,v[,mode])	Changes specific elements of one array by replacing from another array.
putmask(a,mask,values)	Changes elements of an array based on conditional and input values.

place(*arr, mask, vals*)

Changes elements of an array based on conditional and input values.

Similar to `putmask(a, mask, vals)` but the 1D array *vals* has the same number of elements as the non-zero values of *mask*. Inverse of `extract`.

Sets *a*.flat[n] = *values*[n] for each n where *mask*.flat[n] is true.

Parameters

a : array_like

Array to put data into.

mask : array_like

Boolean mask array.

values : array_like, shape(number of non-zero *mask*,)

Values to put into *a*.

See Also:

putmask, put, take

put (*a, ind, v, mode='raise'*)

Changes specific elements of one array by replacing from another array.

Set *a*.flat[n] = *v*[n] for all n in *ind*. If *v* is shorter than *ind*, it will repeat which is different than *a[ind]* = *v*.

Parameters

a : array_like (contiguous)

Target array.

ind : array_like

Target indices, interpreted as integers.

v : array_like

Values to place in *a* 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

If *v* is shorter than *mask* it will be repeated as necessary. In particular *v* can be a scalar or length 1 array. The routine put is the equivalent of the following (although the loop is in C for speed):

```
ind = array(indices, copy=False)
v = array(values, copy=False).astype(a.dtype)
for i in ind: a.flat[i] = v[i]
```

Examples

```
>>> x = np.arange(5)
>>> np.put(x,[0,2,4],[-1,-2,-3])
>>> print x
[-1  1 -2  3 -3]
```

putmask (*a, mask, values*)

Changes elements of an array based on conditional and input values.

Sets *a*.flat[n] = *values*[n] for each n where *mask*.flat[n] is true.

If *values* is not the same size as *a* and *mask* then it will repeat. This gives behavior different from *a[mask]* = *values*.

Parameters

a : array_like

Array to put data into

mask : array_like

Boolean mask array

values : array_like

Values to put

See Also:

`place`, `put`, `take`

Examples

```
>>> a = np.array([10,20,30,40])
>>> mask = np.array([True,False,True,True])
>>> a.putmask([60,70,80,90], mask)
>>> a
array([60, 20, 80, 90])
>>> a = np.array([10,20,30,40])
>>> a[mask]
array([60, 80, 90])
>>> a[mask] = np.array([60,70,80,90])
>>> a
array([60, 20, 70, 80])
>>> a.putmask([10,90], mask)
>>> a
array([10, 20, 10, 90])
>>> np.putmask(a, mask, [60,70,80,90])
```

3.3.4 Iterating over arrays

`ndenumerate`	Multidimensional index iterator.
`ndindex`	Pass in a sequence of integers corresponding to the number of dimensions in the counter. This iterator will then return an N-dimensional counter.
`flatiter`	

class ndenumerate(*arr*)

Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

Parameters

a : ndarray

Input array.

Examples

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

class ndindex(**args*)

Pass in a sequence of integers corresponding to the number of dimensions in the counter. This iterator will then return an N-dimensional counter.

Example: >>> 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)

class flatiter()

3.4 Data type routines

`can_cast`([from,to])	Returns True if cast between data types can occur without losing precision.
`common_type`(*arrays)	Return the inexact scalar type which is most common in a list of arrays.
`obj2sctype`(rep[,default])	

`can_cast`(*from=d1, to=d2*)

Returns True if cast between data types can occur without losing precision.

Parameters

from: data type code :
Data type code to cast from.

to: data type code :
Data type code to cast to.

Returns

out : bool
True if cast can occur without losing precision.

`common_type`(**arrays*)

Return the inexact scalar type which is most common in a list of arrays.

The return type will always be an inexact scalar type, even if all the arrays are integer arrays

Parameters

array1, array2, ... : ndarray
Input arrays.

Returns

out : data type code
Data type code.

See Also:

`dtype`

Examples

```
>>> np.common_type(np.arange(4), np.array([45,6]), np.array([45.0, 6.0]))
<type 'numpy.float64'>
```

`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:

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

Record, one field name 'f1', containing int16:

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

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

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

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

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

Using array-protocol type strings:

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

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

```
>>> np.dtype("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:

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

Subdivide `int16` into 2 `int8`'s, called x and y. 0 and 1 are the offsets in bytes:

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

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

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

Offsets in bytes, here 0 and 25:

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

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 : string or list

comma-separated format descriptions — 'f8, i4, a5' list of format description strings — ['f8', 'i4', 'a5']

names : string or (list or tuple of strings)

comma-separated field names — 'col1, col2, col3' list or tuple of field names

titles : sequence

sequence of title strings or unicode

aligned : bool

align the fields by padding as the C-compiler would

byteorder : :

If specified, all the fields will be changed to the provided byteorder. Otherwise, the default byteorder is used.

Returns

object :

A Python object whose dtype attribute is a data-type.

3.4.2 Data type information

`finfo`(dtype)	Machine limits for floating point types.
`iinfo`(type)	Machine limits for integer types.
`MachAr`	Diagnosing machine parameters.

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.

Attributes

eps : floating point number of the appropriate type

The smallest representable number such that `1.0 + eps != 1.0`.

epsneg : floating point number of the appropriate type

The smallest representable number such that `1.0 - epsneg != 1.0`.

iexp : int

The number of bits in the exponent portion of the floating point representation.

machar : MachAr

The object which calculated these parameters and holds more detailed information.

machep : int

The exponent that yields `eps`.

max : floating point number of the appropriate type

The largest representable number.

maxexp : int

The smallest positive power of the base (2) that causes overflow.

min : floating point number of the appropriate type

The smallest representable number, typically `-max`.

minexp : int

The most negative power of the base (2) consistent with there being no leading 0s in the mantissa.

negep : int

The exponent that yields `epsneg`.

nexp : int

The number of bits in the exponent including its sign and bias.

nmant : int

The number of bits in the mantissa.

precision : int

The approximate number of decimal digits to which this kind of float is precise.

resolution : floating point number of the appropriate type

The approximate decimal resolution of this type, i.e. `10**-precision`.

tiny : floating point number of the appropriate type

The smallest-magnitude usable number.

See Also:

`numpy.lib.machar.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.

class **`iinfo`**(*int_type*)

Machine limits for integer types.

Parameters

type : integer type, dtype, or instance

The kind of integer data type to get information about.

Attributes

min : int

The smallest integer expressible by the type.

max : int

The largest integer expressible by the type.

See Also:

`finfo`
The equivalent for floating point data types.

Examples

With types:

```
>>> ii16 = np.iinfo(np.int16)
>>> ii16.min
-32768
>>> ii16.max
32767
>>> ii32 = np.iinfo(np.int32)
>>> ii32.min
-2147483648
>>> ii32.max
2147483647
```

With instances:

```
>>> ii32 = np.iinfo(np.int32(10))
>>> ii32.min
-2147483648
>>> ii32.max
2147483647
```

class **MachAr** (*float_conv=<type 'float'>, int_conv=<type 'int'>, float_to_float=<type 'float'>, float_to_str=<function <lambda> at 0x139e2938>, title='Python floating point number'*)

Diagnosing machine parameters.

Attributes

ibeta : int

Radix in which numbers are represented.

it : int

Number of base-*ibeta* digits in the floating point mantissa M.

machep : int

Exponent of the smallest (most negative) power of *ibeta* that, added to 1.0, gives something different from 1.0

eps : float

Floating-point number `beta**machep` (floating point precision)

negep : int

Exponent of the smallest power of *ibeta* that, substracted from 1.0, gives something different from 1.0.

epsneg : float

Floating-point number `beta**negep`.

iexp : int

Number of bits in the exponent (including its sign and bias).

minexp : int

Smallest (most negative) power of *ibeta* consistent with there being no leading zeros in the mantissa.

xmin : float

Floating point number `beta**minexp` (the smallest [in magnitude] usable floating value).

maxexp : int

Smallest (positive) power of *ibeta* that causes overflow.

xmax : float

`(1-epsneg) * beta**maxexp` (the largest [in magnitude] usable floating value).

irnd : int

In `range(6)`, information on what kind of rounding is done in addition, and on how underflow is handled.

ngrd : int

Number of 'guard digits' used when truncating the product of two mantissas to fit the representation.

epsilon : float

Same as *eps*.

tiny : float

Same as *xmin*.

huge : float

Same as *xmax*.

precision : float

`- int(-log10(eps))`

resolution : float

`` - 10**(-precision)``

References

3.4.3 Data type testing

`issctype`(rep)	Determines whether the given object represents a numeric array type.
`issubdtype`(arg1,arg2)	Returns True if first argument is a typecode lower/equal in type hierarchy.
`issubsctype`(arg1,arg2)	
`issubclass_`(arg1,arg2)	
`find_common_type`(array_types,scalar_types)	Determine common type following standard coercion rules

`issctype`(*rep*)

Determines whether the given object represents a numeric array type.

`issubdtype`(*arg1, arg2*)

Returns True if first argument is a typecode lower/equal in type hierarchy.

Parameters

arg1 : dtype_like

dtype or string representing a typecode.

arg2 : dtype_like

dtype or string representing a typecode.

See Also:

`numpy.core.numerictypes`

Overview of numpy type hierarchy.

Examples

```
>>> np.issubdtype('S1', str)
True
>>> np.issubdtype(np.float64, np.float32)
False
```

`issubsctype`(*arg1, arg2*)

`issubclass_`(*arg1, arg2*)

`find_common_type`(*array_types, scalar_types*)

Determine common type following standard coercion rules

Parameters

array_types : sequence

A list of dtype convertible objects representing arrays

scalar_types : sequence

A list of dtype convertible objects representing scalars

Returns

datatype : dtype

The common data-type which is the maximum of the array_types ignoring the scalar_types unless the maximum of the scalar_types is of a different kind.

If the kinds is not understood, then None is returned.

See Also:

`dtype`

3.4.4 Miscellaneous

`typename`(char)	Return a description for the given data type code.
`sctype2char`(sctype)	
`mintypecode`(typechars[,typeset,default])	Return a minimum data type character from typeset that handles all typechars given

`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:

`typecodes`, `dtype`

`sctype2char`(*sctype*)

mintypecode (*typechars, typeset='GDFgdf', default='d'*)

Return a minimum data type character from typeset that handles all typechars given

The returned type character must be the smallest size such that an array of the returned type can handle the data from an array of type t for each t in typechars (or if typechars is an array, then its dtype.char).

If the typechars does not intersect with the typeset, then default is returned.

If t in typechars is not a string then t=asarray(t).dtype.char is applied.

3.5 Input and output

3.5.1 NPZ files

`load`(file[,mmap_mode])	Load a pickled, `.npy`, or `.npz` binary file.
`save`(file,arr)	Save an array to a binary file in NumPy format.
`savez`(file,*args,**kwds)	Save several arrays into an .npz file format which is a zipped-archive of arrays

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.

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.

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:

```
>>> 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:

```
>>> 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 format.

Parameters
f : 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.
x : array_like
Array data.

Examples

```
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
```

```
>>> x = np.arange(10)
>>> np.save(outfile, x)
```

```
>>> outfile.seek(0)
>>> np.load(outfile)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

savez (*file, *args, **kwds*)
Save several arrays into an .npz file format which is a zipped-archive of arrays

If keyword arguments are given, then filenames are taken from the keywords. If arguments are passed in with no keywords, then stored file names are arr_0, arr_1, etc.

Parameters
file : string
File name of .npz file.
args : Arguments
Function arguments.
kwds : Keyword arguments
Keywords.

3.5.2 Text files

`loadtxt`(fname[,dtype,comments,...])	Load data from a text file.
`savetxt`(fname,X[,fmt,delimiter])	Save an array to file.
`fromregex`(file,regexp,dtype)	Construct an array from a text file, using regular-expressions parsing.
`fromstring`(string[,dtype,count,sep])	Return a new 1d array initialized from raw binary or text data in string.
`ndarray.tofile`(fid[,sep,format])	Write array to a file as text or binary.
`ndarray.tolist`()	Return the array as a possibly nested list.

`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 string

File or filename to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed.

dtype : data-type

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 : string, optional

The character used to indicate the start of a comment.

delimiter : string, optional

The string used to separate values. By default, this is any whitespace.

converters : {}

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

Skip the first *skiprows* lines.

usecols : sequence

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

If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`

Returns

out : ndarray

Data read from the text file.

See Also:

`scipy.io.loadmat`

reads Matlab(R) data files

Examples

```
>>> 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={'names': ('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 file.

Parameters

fname : filename or a file handle

If the filename ends in .gz, the file is automatically saved in compressed gzip format. The load() command understands gzipped files transparently.

X : array_like

Data.

fmt : string or sequence of strings

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.

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 is not an exhaustive specification.

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
```

fromregex (*file, regexp, dtype*)

Construct an array from a text file, using regular-expressions parsing.

Array is constructed from all matches of the regular expression in the file. Groups in the regular expression are converted to fields.

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 dtype list

Dtype for the structured array

Examples

```
>>> f = open('test.dat', 'w')
>>> f.write("1312 foo\n1534  bar\n444   qux")
>>> f.close()
>>> np.fromregex('test.dat', r"(\d+)\s+(...)",
...              [('num', np.int64), ('key', 'S3')])
array([(1312L, 'foo'), (1534L, 'bar'), (444L, 'qux')],
      dtype=[('num', '<i8'), ('key', '|S3')])
```

fromstring (*string, dtype=float, count=-1, sep=''*)

Return a new 1d 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('\x01\x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
>>> np.fromstring('\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
```

Invalid inputs:

```
>>> 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
>>> 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 (*fid, sep="", format="%s"*)

Write array to a file as text or binary.

Data is always written in 'C' order, independently of the order of *a*. The data produced by this method can be recovered by using the function fromfile().

This is a convenience function for quick storage of array data. Information on endianess 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 endianess. Some of these problems can be overcome by outputting the data as text files at the expense of speed and file size.

Parameters

fid : file or string

An open file object or a string containing a filename.

sep : string

Separator between array items for text output. If "" (empty), a binary file is written, equivalently to file.write(a.tostring()).

format : string

Format string for text file output. Each entry in the array is formatted to text by converting it to the closest Python type, and using "format" % item.

tolist ()

Return the array as a possibly nested list.

Return a copy of the array data as a hierarchical Python list. Data items are converted to the nearest compatible Python type.

Parameters
: **none** :

Returns
: **y** : list
: The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

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

3.5.3 String formatting

`array_repr`(arr[,max_line_width,...])	Return the string representation of an array.
`array_str`(a[,max_line_width,...])	Return a string representation of an array.

`array_repr`(*arr, max_line_width=None, precision=None, suppress_small=None*)

Return the string representation of an array.

Parameters
: **arr** : ndarray
: Input array.
: **max_line_width** : int
: The maximum number of columns the string should span. Newline characters splits the string appropriately after array elements.
: **precision** : int
: Floating point precision.
: **suppress_small** : bool
: Represent very small numbers as zero.

Returns
: **string** : str
: The string representation of an array.

Examples

```
>>> np.array_repr(np.array([1,2]))
'array([1, 2])'
>>> np.array_repr(np.ma.array([0.]))
'MaskedArray([ 0.])'
>>> np.array_repr(np.array([], np.int32))
'array([], dtype=int32)'
```

array_str(*a, max_line_width=None, precision=None, suppress_small=None*)

Return a string representation of an array.

Parameters

a : ndarray

Input array.

max_line_width : int, optional

Inserts newlines if text is longer than *max_line_width*.

precision : int, optional

If *a* is float, *precision* sets loating point precision.

suppress_small : boolean, optional

Represent very small numbers as zero.

See Also:

`array2string`, `array_repr`

Examples

```
>>> np.array_str(np.arange(3))
>>> '[0 1 2]'
```

3.5.4 Memory mapping files

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

class memmap()

Create a memory-map to an array stored in a 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 : string or file-like object

The file name or file object to be used as the array data buffer.

dtype : data-type, optional

The data-type used to interpret the file contents. Default is *uint8*

mode : {'r+', 'r', 'w+', 'c'}, optional

The file is opened in this mode:

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

Default is 'r+'.

offset : integer, optional

In the file, array data starts at this offset. *offset* 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 defaullt order is 'C'.

Methods

close :

Close the memmap file.

flush :

Flush any changes in memory to file on disk. When you delete a memmap object, flush is called first to write changes to disk before removing the object.

Notes

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.

Examples

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

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

```
>>> from tempfile import mkdtemp
>>> import os.path as path
>>> filename = path.join(mkdtemp(), 'newfile.dat')
```

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

```
>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
>>> fp
memmap([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]], dtype=float32)
```

Write data to memmap array:

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

Deletion flushes memory changes to disk before removing the object:

```
>>> del fp
```

Load the memmap and verify data was stored:

```
>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> newfp
memmap([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
```

Read-only memmap:

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

Cannot assign to read-only, obviously:

```
>>> fpr[0, 3] = 56
Traceback (most recent call last):
    ...
RuntimeError: array is not writeable
```

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)
```

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 : string, optional

String representation of floating point not-a-number (default nan).

infstr : string, optional

String representation of floating point infinity (default inf).

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 : string
- infstr : string

See Also:

`set_printoptions`
parameter descriptions

`set_string_function` (*f, repr=1*)
Set a Python function to be used when pretty printing arrays.

Parameters
f : Python function
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.
repr : int
Unknown.

Examples

```
>>> def pprint(arr):
...     return 'HA! - What are you going to do now?'
...
>>> np.set_string_function(pprint)
>>> a = np.arange(10)
>>> a
HA! - What are you going to do now?
>>> print a
[0 1 2 3 4 5 6 7 8 9]
```

3.5.6 Base-n representations

`binary_repr`(num[,width])	Return the binary representation of the input number as a string.
`base_repr`(number[,base,padding])	Return a string representation of a number in the given base system.

`binary_repr` (*num, width=None*)
Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two's complement of the number is returned, with respect to that width.

In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two's-complement system can represent every integer in the range -2^{N-1} to $+2^{N-1}-1$.

Parameters
num : int
Only an integer decimal number can be used.
width : int, optional

[1] Wikipedia, "Two's complement", http://en.wikipedia.org/wiki/Two's_complement

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`

Notes

binary_repr is equivalent to using *base_repr* with base 2, but about 25x faster.

References

Examples

```
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

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

```
>>> np.binary_repr(-3, width=4)
'1101'
```

`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

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.

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

```
>>> np.base_repr(3, 5)
'3'
>>> np.base_repr(6, 5)
'11'
>>> np.base_repr(7, 5, padding=3)
'00012'
```

3.5.7 Data sources

DataSource	A generic data source file (file, http, ftp, ...).

class DataSource(*destpath='.'*)

A generic data source file (file, http, ftp, ...).

DataSources could 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.

Methods:

- exists : test if the file exists locally or remotely
- abspath : get absolute path of the file in the DataSource directory
- open : open the file

Example URL DataSource:

```
# Initialize DataSource with a local directory, default is os.curdir.
ds = DataSource('/home/guido')

# Open remote file.
# File will be downloaded and opened from here:
#     /home/guido/site/xyz.txt
ds.open('http://fake.xyz.web/site/xyz.txt')
```

Example using DataSource for temporary files:

```
# Initialize DataSource with 'None' for the local directory.
ds = DataSource(None)

# Open local file.
# Opened file exists in a temporary directory like:
#     /tmp/tmpUnhcvM/foobar.txt
# Temporary directories are deleted when the DataSource is deleted.
ds.open('/home/guido/foobar.txt')
```

***Notes*:**

BUG

[URLs require a scheme string ('http://') to be used.] www.google.com will fail.

```
>>> repos.exists('www.google.com/index.html')
False

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

3.6 Fourier transforms (`numpy.fft`)

3.6.1 1-dimensional

`fft`(a[,n,axis])	Compute the one dimensional fft on a given axis.
`ifft`(a[,n,axis])	Compute the one-dimensonal inverse fft along an axis.

`fft`(*a, n=None, axis=-1*)

Compute the one dimensional fft on a given axis.

Return the n point discrete Fourier transform of a. n defaults to the length of a. If n is larger than the length of a, then a will be zero-padded to make up the difference. If n is smaller than the length of a, only the first n items in a will be used.

Parameters

a : array

input array

n : int

length of the fft

axis : int

axis over which to compute the fft

Notes

The packing of the result is “standard”: If A = fft(a, n), then A[0] contains the zero-frequency term, A[1:n/2+1] contains the positive-frequency terms, and A[n/2+1:] contains the negative-frequency terms, in order of decreasingly negative frequency. So for an 8-point transform, the frequencies of the result are [0, 1, 2, 3, 4, -3, -2, -1].

This is most efficient for n a power of two. This also stores a cache of working memory for different sizes of fft’s, so you could theoretically run into memory problems if you call this too many times with too many different n’s.

`ifft`(*a, n=None, axis=-1*)

Compute the one-dimensonal inverse fft along an axis.

Return the *n* point inverse discrete Fourier transform of *a*. The length *n* defaults to the length of *a*. If *n* is larger than the length of *a*, then *a* will be zero-padded to make up the difference. If *n* is smaller than the length of *a*, then *a* will be truncated to reduce its size.

Parameters

a : array_like

Input array.

n : int, optional

Length of the fft.

axis : int, optional

Axis over which to compute the inverse fft.

See Also:

`fft`

Notes

The input array is expected to be packed the same way as the output of fft, as discussed in the fft documentation.

This is the inverse of fft: ifft(fft(a)) == a within numerical accuracy.

This is most efficient for *n* a power of two. This also stores a cache of working memory for different sizes of fft's, so you could theoretically run into memory problems if you call this too many times with too many different *n* values.

3.6.2 2-dimensional

`fft2`(a[,s,axes,-1))	Compute the 2-D FFT of an array.
`ifft2`(a[,s,axes,-1))	Compute the inverse 2d fft of an array.

`fft2` (*a, s=None, axes=(-2, -1)*)

Compute the 2-D FFT of an array.

Parameters

a : array_like

Input array. The rank (dimensions) of *a* must be 2 or greater.

s : shape tuple

Shape of the FFT.

axes : sequence of 2 ints

The 2 axes over which to compute the FFT. The default is the last two axes (-2, -1).

Notes

This is really just `fftn` with different default behavior.

`ifft2` (*a, s=None, axes=(-2, -1)*)

Compute the inverse 2d fft of an array.

Parameters

a : array

input array

s : sequence (int)

shape of the ifft

axis : int

axis over which to compute the ifft

Notes

This is really just ifftn with different default behavior.

3.6.3 N-dimensional

`fftn`(a[,s,axes])	Compute the N-dimensional Fast Fourier Transform.
`ifftn`(a[,s,axes])	Compute the inverse of fftn.

`fftn` (*a, s=None, axes=None*)

Compute the N-dimensional Fast Fourier Transform.

Parameters

a : array_like

Input array.

s : sequence of ints

Shape of each axis of the input (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.

axes : tuple of int

Axes over which to compute the FFT.

Notes

Analogously to *fft*, the term for zero frequency in all axes is in the low-order corner, while the term for the Nyquist frequency in all axes is in the middle.

If neither *s* nor *axes* is specified, the transform is taken along all axes. If *s* is specified and *axes* is not, the last `len(s)` axes are used. If *axes* is specified and *s* is not, the input shape along the specified axes is used. If *s* and *axes* are both specified and are not the same length, an exception is raised.

`ifftn`(*a, s=None, axes=None*)

Compute the inverse of fftn.

Parameters

a : array

input array

s : sequence (int)

shape of the ifft

axis : int

axis over which to compute the ifft

Notes

The n-dimensional ifft of a. s is a sequence giving the shape of the input an result along the transformed axes, as n for fft. Results are packed analogously to fft: the term for zero frequency in all axes is in the low-order corner, while the term for the Nyquist frequency in all axes is in the middle.

If neither s nor axes is specified, the transform is taken along all axes. If s is specified and axes is not, the last len(s) axes are used. If axes are specified and s is not, the input shape along the specified axes is used. If s and axes are both specified and are not the same length, an exception is raised.

3.6.4 Hermite symmetric

`hfft`(a[,n,axis])	Compute the fft of a signal which spectrum has Hermitian symmetry.
`ihfft`(a[,n,axis])	Compute the inverse fft of a signal whose spectrum has Hermitian symmetry.

`hfft`(*a, n=None, axis=-1*)

Compute the fft of a signal which spectrum has Hermitian symmetry.

Parameters

a : array

input array

n : int

length of the hfft

axis : int

axis over which to compute the hfft

See Also:

`rfft`, `ihfft`

Notes

These 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 hermite_fft 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.

ihfft (*a, n=None, axis=-1*)

Compute the inverse fft of a signal whose spectrum has Hermitian symmetry.

Parameters

a : array_like
  Input array.

n : int, optional
  Length of the ihfft.

axis : int, optional
  Axis over which to compute the ihfft.

See Also:

rfft, hfft

Notes

These 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 hermite_fft 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.

3.6.5 Real-valued

rfft(a[,n,axis])	Compute the one-dimensional fft for real input.
irfft(a[,n,axis])	Compute the one-dimensional inverse fft for real input.
rfft2(a[,s,axes,-1))	Compute the 2-dimensional fft of a real array.
irfft2(a[,s,axes,-1))	Compute the 2-dimensional inverse fft of a real array.
rfftn(a[,s,axes])	Compute the n-dimensional fft of a real array.
irfftn(a[,s,axes])	Compute the n-dimensional inverse fft of a real array.

rfft (*a, n=None, axis=-1*)

Compute the one-dimensional fft for real input.

Return the n point discrete Fourier transform of the real valued array a. n defaults to the length of a. n is the length of the input, not the output.

Parameters

a : array
  input array with real data type

n : int
  length of the fft

axis : int
  axis over which to compute the fft

Notes

The returned array will be the nonnegative frequency terms of the Hermite-symmetric, complex transform of the real array. So for an 8-point transform, the frequencies in the result are [0, 1, 2, 3, 4]. The first term will be real, as will the last if n is even. The negative frequency terms are not needed because they are the complex conjugates of the positive frequency terms. (This is what I mean when I say Hermite-symmetric.)

This is most efficient for n a power of two.

`irfft` (*a, n=None, axis=-1*)

Compute the one-dimensional inverse fft for real input.

Parameters

a : array

Input array with real data type.

n : int

Length of the fft.

axis : int

Axis over which to compute the fft.

See Also:

`rfft`

Notes

Return the real valued *n* point inverse discrete Fourier transform of *a*, where *a* contains the nonnegative frequency terms of a Hermite-symmetric sequence. *n* is the length of the result, not the input. If *n* is not supplied, the default is 2*(len(*a*)-1). If you want the length of the result to be odd, you have to say so.

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_resamp = irfft(rfft(a), m).

Within numerical accuracy `irfft` is the inverse of `rfft`:

```
irfft(rfft(a), len(a)) == a
```

`rfft2` (*a, s=None, axes=(-2, -1)*)

Compute the 2-dimensional fft of a real array.

Parameters

a : array (real)

input array

s : sequence (int)

shape of the fft

axis : int

axis over which to compute the fft

Notes

The 2-D fft of the real valued array a. This is really just rfftn with different default behavior.

`irfft2` (*a, s=None, axes=(-2, -1)*)

Compute the 2-dimensional inverse fft of a real array.

Parameters

a : array (real)

input array

s : sequence (int)

shape of the inverse fft

axis : int

axis over which to compute the inverse fft

Notes

This is really irfftn with different default.

`rfftn`(*a, s=None, axes=None*)

Compute the n-dimensional fft of a real array.

Parameters

a : array (real)

input array

s : sequence (int)

shape of the fft

axis : int

axis over which to compute the fft

Notes

A real transform as rfft is performed along the axis specified by the last element of axes, then complex transforms as fft are performed along the other axes.

`irfftn`(*a, s=None, axes=None*)

Compute the n-dimensional inverse fft of a real array.

Parameters

a : array (real)

input array

s : sequence (int)

shape of the inverse fft

axis : int

axis over which to compute the inverse fft

Notes

The transform implemented in ifftn is applied along all axes but the last, then the transform implemented in irfft is performed along the last axis. As with irfft, the length of the result along that axis must be specified if it is to be odd.

3.6.6 Helper routines

`fftfreq`(n[,d])	Discrete Fourier Transform sample frequencies.
`fftshift`(x[,axes])	Shift zero-frequency component to center of spectrum.
`ifftshift`(x[,axes])	Inverse of fftshift.

`fftfreq`(*n, d=1.0*)

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*.

```
f = [0,1,...,n/2-1,-n/2,...,-1]/(d*n)         if n is even
f = [0,1,...,(n-1)/2,-(n-1)/2,...,-1]/(d*n)   if n is odd
```

Parameters

n : int

Window length.

d : scalar

Sample spacing.

Returns

out : ndarray, shape(*n*,)

Sample frequencies.

Examples

```
>>> signal = np.array([-2.,  8., -6.,  4.,  1., 0.,  3.,  5.])
>>> fourier = np.fft.fft(signal)
>>> n = len(signal)
>>> 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 zero-frequency component to center of spectrum.

This function swaps half-spaces for all axes listed (defaults to all). If len(x) is even then the Nyquist component is y[0].

Parameters

x : array_like

Input array.

axes : int or shape tuple, optional

Axes over which to shift. Default is None which shifts all axes.

See Also:

`ifftshift`

ifftshift (*x, axes=None*)

Inverse of fftshift.

Parameters

x : array_like

Input array.

axes : int or shape tuple, optional

Axes over which to calculate. Defaults to None which is over all axes.

See Also:

`fftshift`

3.7 Linear algebra (`numpy.linalg`)

3.7.1 Matrix and vector products

`dot`(a,b)	Dot product of two arrays.
`vdot`(a,b)	Return the dot product of two vectors.
`inner`(a,b)	Inner product of two arrays.
`outer`(a,b)	Returns the outer product of two vectors.
`tensordot`(a,b[,axes])	Returns the tensor dot product for (ndim >= 1) arrays along an axes.
`linalg.matrix_power`(M,n)	Raise a square matrix to the (integer) power n.
`kron`(a,b)	Kronecker product of two arrays.

dot (*a, b*)

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of *a* and the second-to-last of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters

a : array_like

First argument.

b : array_like

Second argument.

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)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

vdot(*a, b*)

Return the dot product of two vectors.

The vdot(*a*, *b*) function handles complex numbers differently than dot(*a*, *b*). If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

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

```
>>> 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(*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

For vectors (1-D arrays) it computes the ordinary inner-product:

```
np.inner(a, b) = sum(a[:]*b[:])
```

More generally, if *ndim(a) = r > 0* and *ndim(b) = s > 0*:

```
np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))
```

or explicitly:

```
np.inner(a, b)[i0,...,ir-1,j0,...,js-1]
     = sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])
```

In addition *a* or *b* may be scalars, in which case:

```
np.inner(a,b) = a*b
```

Examples

Ordinary inner product for vectors:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

outer (*a, b*)

Returns the outer product of two vectors.

Given two vectors, `[a0, a1, ..., aM]` and `[b0, b1, ..., bN]`, the outer product becomes:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

Parameters

a : array_like, shaped (M,)

First input vector. If either of the input vectors are not 1-dimensional, they are flattened.

b : array_like, shaped (N,)

Second input vector.

Returns

out : ndarray, shaped (M, N)

`out[i, j] = a[i] * b[j]`

Notes

The outer product of vectors is a special case of the Kronecker product.

Examples

```
>>> 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)
```

tensordot (*a, b, axes=2*)

Returns the tensor dot product for (ndim >= 1) arrays along an axes.

The first element of the sequence determines the axis or axes in *a* to sum over, and the second element in *axes* argument sequence determines the axis or axes in *b* to sum over.

Parameters

a : array_like

Input array.

b : array_like

Input array.

axes : shape tuple

Axes to be summed over.

See Also:

dot

Notes

r_{xxx, yyy} = sum_k a_{xxx,k} b_{k,yyy}

When there is more than one axis to sum over, the corresponding arguments to axes should be sequences of the same length with the first axis to sum over given first in both sequences, the second axis second, and so forth.

If the *axes* argument is an integer, N, then the last N dimensions of *a* and first N dimensions of *b* are summed over.

Examples

```
>>> 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...
>>> c = np.zeros((5,2))
>>> for i in range(5):
...   for j in range(2):
...     for k in range(3):
...       for n in range(4):
...         c[i,j] += a[k,n,i] * b[n,k,j]
```

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 type as M is returned. If n<0, the inverse is computed and raised to the exponent.

Parameters

M : array_like

Must be a square array (that is, of dimension two and with equal sizes).

n : integer

The exponent can be any integer or long integer, positive negative or zero.

Returns

M to the power n :

The return value is a an array the same shape and size as M; if the exponent was positive or zero then the type of the elements is the same as those of M. If the exponent was negative the elements are floating-point.

Raises

LinAlgException :

If the matrix is not numerically invertible, an exception is raised.

See Also:

`The, operator.`

Examples

```
>>> np.linalg.matrix_power(np.array([[0,1],[-1,0]]),10)
array([[-1,  0],
       [ 0, -1]])
```

kron (*a, b*)

Kronecker product of two arrays.

Computes the Kronecker product, a composite array made of blocks of the second array scaled by the first.

Parameters

a, b : array_like

Returns

out : ndarray

See Also:

outer

The outer product

Notes

The function assumes that the number of dimenensions of *a* and *b* are the same, if necessary prepending the smallest with ones. If *a.shape = (r0,r1,..,rN)* and *b.shape = (s0,s1,...,sN)*, the Kronecker product has shape *(r0*s0, r1*s1, ..., rN*SN)*. The elements are products of elements from *a* and *b*, organized explicitly by:

```
kron(a,b)[k0,k1,...,kN] = a[i0,i1,...,iN] * b[j0,j1,...,jN]
```

where:

```
kt = it * st + jt,  t = 0,...,N
```

In the common 2-D case (N=1), the block structure can be visualized:

```
[[ a[0,0]*b,   a[0,1]*b,  ... , a[0,-1]*b  ],
 [  ...                              ...   ],
 [ a[-1,0]*b,  a[-1,1]*b, ... , a[-1,-1]*b ]]
```

Examples

```
>>> np.kron([1,10,100], [5,6,7])
array([  5,   6,   7,  50,  60,  70, 500, 600, 700])
>>> np.kron([5,6,7], [1,10,100])
array([  5,  50, 500,   6,  60, 600,   7,  70, 700])
```

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

```
>>> a = np.arange(100).reshape((2,5,2,5))
>>> b = np.arange(24).reshape((2,3,4))
>>> c = np.kron(a,b)
>>> c.shape
(2, 10, 6, 20)
>>> I = (1,3,0,2)
>>> J = (0,2,1)
>>> J1 = (0,) + J             # extend to ndim=4
>>> S1 = (1,) + b.shape
>>> K = tuple(np.array(I) * np.array(S1) + np.array(J1))
>>> C[K] == A[I]*B[J]
True
```

3.7.2 Decompositions

`linalg.cholesky`(a)	Cholesky decomposition.
`linalg.qr`(a[,mode])	Compute QR decomposition of a matrix.
`linalg.svd`(a[,full_matrices,compute_uv])	Singular Value Decomposition.

`cholesky`(*a*)

Cholesky decomposition.

Return the Cholesky decomposition, $A = LL^*$ of a Hermitian positive-definite matrix A.

Parameters

a : array_like, shape (M, M)

Hermitian (symmetric, if it is real) and positive definite input matrix.

Returns

L : array_like, shape (M, M)

Lower-triangular Cholesky factor of A.

Raises

LinAlgError :

If the decomposition fails.

Notes

The Cholesky decomposition is often used as a fast way of solving

$$A\mathbf{x} = \mathbf{b}.$$

First, we solve for $\mathbf{y}$ in

$$L\mathbf{y} = \mathbf{b},$$

and then for $\mathbf{x}$ in

$$L^*\mathbf{x} = \mathbf{y}.$$

Examples

```
>>> A = np.array([[1,-2j],[2j,5]])
>>> 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())
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
```

qr (*a, mode='full'*)

Compute QR decomposition of a matrix.

Calculate the decomposition $A = QR$ where Q is orthonormal and R upper triangular.

Parameters

a : array_like, shape (M, N)

Matrix to be decomposed

mode : {'full', 'r', 'economic'}

Determines what information is to be returned. 'full' is the default. Economic mode is slightly faster if only R is needed.

Returns

mode = 'full' :

Q : double or complex array, shape (M, K)

R : double or complex array, shape (K, N)

Size K = min(M, N)

mode = 'r' :

R : double or complex array, shape (K, N)

mode = 'economic' :

A2 : double or complex array, shape (M, N)

The diagonal and the upper triangle of A2 contains R, while the rest of the matrix is undefined.

If a is a matrix, so are all the return values. :

Raises LinAlgError if decomposition fails :

Notes

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dorgqr, and zungqr.

Examples

```
>>> a = np.random.randn(9, 6)
>>> q, r = np.linalg.qr(a)
>>> np.allclose(a, np.dot(q, r))
True
```

```
>>> r2 = np.linalg.qr(a, mode='r')
>>> r3 = np.linalg.qr(a, mode='economic')
>>> np.allclose(r, r2)
True
>>> np.allclose(r, np.triu(r3[:6,:6], k=0))
True
```

svd(*a, full_matrices=1, compute_uv=1*)

Singular Value Decomposition.

Factorizes the matrix *a* into two unitary matrices, `U` and `Vh`, and a 1-dimensional array of singular values, `s` (real, non-negative), such that `a == U S Vh`, where `S` is the diagonal matrix `np.diag(s)`.

Parameters

a : array_like, shape (M, N)

Matrix to decompose

full_matrices : boolean, optional

If True (default), `U` and `Vh` are shaped `(M,M)` and `(N,N)`. Otherwise, the shapes are `(M,K)` and `(K,N)`, where `K = min(M,N)`.

compute_uv : boolean

Whether to compute `U` and `Vh` in addition to `s`. True by default.

Returns

U : ndarray, shape (M, M) or (M, K) depending on *full_matrices*

Unitary matrix.

s : ndarray, shape (K,) where `K = min(M, N)`

The singular values, sorted so that `s[i] >= s[i+1]`.

Vh : ndarray, shape (N,N) or (K,N) depending on *full_matrices*

Unitary matrix.

Raises

LinAlgError :

If SVD computation does not converge.

Notes

If *a* is a matrix (in contrast to an ndarray), then so are all the return values.

Examples

```
>>> 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 eigenvalues and right eigenvectors of an array.
`linalg.eigh`(a[,UPLO])	Eigenvalues and eigenvectors of a Hermitian or real symmetric matrix.
`linalg.eigvals`(a)	Compute the eigenvalues of a general matrix.
`linalg.eigvalsh`(a[,UPLO])	Compute the eigenvalues of a Hermitean or real symmetric matrix.

eig(*a*)

Compute eigenvalues and right eigenvectors of an array.

Parameters

a : array_like, shape (M, M)

A complex or real 2-D array.

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 matrices.

v : ndarray, shape (M, M)

The normalized eigenvector corresponding to the eigenvalue `w[i]` is the column `v[:,i]`.

Raises

LinAlgError :

If the eigenvalue computation does not converge.

See Also:

eigvalsh
eigenvalues of symmetric or Hemitiean arrays.

eig
eigenvalues and right eigenvectors for non-symmetric arrays

eigvals
eigenvalues of non-symmetric array.

Notes

This is a simple interface to the LAPACK routines dgeev and zgeev that compute the eigenvalues and eigenvectors of general real and complex arrays respectively.

The number *w* is an eigenvalue of a if there exists a vector *v* satisfying the equation `dot(a,v) = w*v`. Alternately, if *w* is a root of the characteristic equation `det(a - w[i]*I) = 0`, where *det* is the determinant and *I* is the identity matrix. The arrays *a*, *w*, and *v* satisfy the equation `dot(a,v[i]) = w[i]*v[:,i]`.

The array *v* of eigenvectors may not be of maximum rank, that is, some of the columns may be dependent, although roundoff error may obscure that fact. If the eigenvalues are all different, then theoretically the eigenvectors are independent. Likewise, the matrix of eigenvectors is unitary if the matrix *a* is normal, i.e., if `dot(a, a.H) = dot(a.H, a)`.

The left and right eigenvectors are not necessarily the (Hermitian) transposes of each other.

eigh(*a, UPLO='L'*)

Eigenvalues and eigenvectors of a Hermitian or real symmetric matrix.

Parameters

a : array_like, shape (M, M)

A complex Hermitian or symmetric real matrix.

UPLO : {'L', 'U'}, optional

Specifies whether the calculation is done with data from the lower triangular part of *a* ('L', default) or the upper triangular part ('U').

Returns

w : ndarray, shape (M,)

The eigenvalues. The eigenvalues are not necessarily ordered.

v : ndarray, shape (M, M)

The normalized eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].

Raises

LinAlgError :

If the eigenvalue computation does not converge.

See Also:

`eigvalsh`
eigenvalues of symmetric or Hemitiean arrays.

`eig`
eigenvalues and right eigenvectors for non-symmetric arrays

`eigvals`
eigenvalues of non-symmetric array.

Notes

A simple interface to the LAPACK routines dsyevd and zheevd that compute the eigenvalues and eigenvectors of real symmetric and complex Hermitian arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix. The eigenvalues of real symmetric or complex Hermitean matrices are always real. The array v of eigenvectors is unitary and a, w, and v satisfy the equation dot(a,v[i]) = w[i]*v[:,i].

eigvals(*a*)

Compute the eigenvalues of a general matrix.

Parameters

a : array_like, shape (M, M)

A complex or real matrix whose eigenvalues and eigenvectors 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 Hemitiean arrays.

`eigh`
eigenvalues and eigenvectors of symmetric/Hermitean arrays.

Notes

This is a simple interface to the LAPACK routines dgeev and zgeev that sets the flags to return only the eigenvalues of general real and complex arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix.

`eigvalsh`(*a, UPLO='L'*)

Compute the eigenvalues of a Hermitean or real symmetric matrix.

Parameters

a : array_like, shape (M, M)

A complex or real matrix whose eigenvalues and eigenvectors will be computed.

UPLO : {'L', 'U'}, optional

Specifies whether the calculation is done with data from the lower triangular part of *a* ('L', default) or the upper triangular part ('U').

Returns

w : ndarray, shape (M,)

The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered.

Raises

LinAlgError :

If the eigenvalue computation does not converge.

See Also:

`eigh`
eigenvalues and eigenvectors of symmetric/Hermitean arrays.

`eigvals`
eigenvalues of general real or complex arrays.

`eig`
eigenvalues and eigenvectors of general real or complex arrays.

Notes

This is a simple interface to the LAPACK routines dsyevd and zheevd that sets the flags to return only the eigenvalues of real symmetric and complex Hermetian arrays respectively.

The number w is an eigenvalue of a if there exists a vector v satisfying the equation dot(a,v) = w*v. Alternately, if w is a root of the characteristic equation det(a - w[i]*I) = 0, where det is the determinant and I is the identity matrix.

3.7.4 Norms and other numbers

`linalg.norm`(x[,ord])	Matrix or vector norm.
`linalg.cond`(x[,p])	Compute the condition number of a matrix.
`linalg.det`(a)	Compute the determinant of an array.
`trace`(a[,offset,axis1,axis2,...])	Return the sum along diagonals of the array.

norm(*x, ord=None*)

Matrix or vector norm.

Parameters

x : array_like, shape (M,) or (M, N)

Input array.

ord : {int, 1, -1, 2, -2, inf, -inf, 'fro'}

Order of the norm (see table under `Notes`).

Returns

n : float

Norm of the matrix or vector

Notes

For values ord < 0, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for numerical purposes.

The following norms can be calculated:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

cond(*x, p=None*)

Compute the condition number of a matrix.

The condition number of *x* is the norm of *x* times the norm of the inverse of *x*. The norm can be the usual L2 (root-of-sum-of-squares) norm or a number of other matrix norms.

Parameters

x : array_like, shape (M, N)

The matrix whose condition number is sought.

p : {None, 1, -1, 2, -2, inf, -inf, 'fro'}

Order of the norm:

p	norm for matrices
None	2-norm, computed directly using the SVD
'fro'	Frobenius norm
inf	max(sum(abs(x), axis=1))
-inf	min(sum(abs(x), axis=1))
1	max(sum(abs(x), axis=0))
-1	min(sum(abs(x), axis=0))
2	2-norm (largest sing. value)
-2	smallest singular value

Returns

c : float

The condition number of the matrix. May be infinite.

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:

```
>>> 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, returns the sum along the diagonal of self 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 trace 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 whis the diagonals are taken.

offset : integer, optional

Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to main diagonal.

axis1 : integer, 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.

axis2 : integer, 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.

dtype : 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 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 : array, optional

Array into which the sum can be 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, a 0-d array containing the diagonal is returned. If a has larger dimensions, then an array of diagonals is returned.

Examples

```
>>> np.trace(np.eye(3))
3.0
>>> a = np.arange(8).reshape((2,2,2))
>>> np.trace(a)
array([6, 8])
```

3.7.5 Solving equations and inverting matrices

`linalg.solve`(a,b)	Solve the equation `a x = b` for `x`.
`linalg.tensorsolve`(a,b[,axes])	Solve the tensor equation a x = b for x
`linalg.lstsq`(a,b[,rcond])	Return the least-squares solution to an equation.
`linalg.inv`(a)	Compute the inverse of a matrix.
`linalg.pinv`(a[,rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
`linalg.tensorinv`(a[,ind])	Find the 'inverse' of a N-d array

solve(*a, b*)

Solve the equation `a x = b` for `x`.

Parameters

a : array_like, shape (M, M)

Input equation coefficients.

b : array_like, shape (M,)

Equation target values.

Returns

x : array, shape (M,)

Raises

LinAlgError :

If *a* is singular or not square.

Examples

Solve the system of equations `3 * x0 + x1 = 9` and `x0 + 2 * x1 = 8`:

```
>>> 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:

```
>>> (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, similarly as in tensordot(a, x, axes=len(b.shape)).

Parameters

a : array_like, shape b.shape+Q

Coefficient tensor. Shape Q of the rightmost indices of a must be such that a is 'square', ie., prod(Q) == prod(b.shape).

b : array_like, any shape

Right-hand tensor.

axes : tuple of integers

Axes in a to reorder to the right, before inversion. If None (default), no reordering is done.

Returns

x : array, shape Q

Examples

```
>>> a = np.eye(2*3*4)
>>> a.shape = (2*3,4,  2,3,4)
>>> b = np.random.randn(2*3,4)
>>> x = np.linalg.tensorsolve(a, b)
>>> x.shape
(2, 3, 4)
>>> np.allclose(np.tensordot(a, x, axes=3), b)
True
```

lstsq(*a, b, rcond=-1*)

Return the least-squares solution to an equation.

Solves the equation *a x = b* by computing a vector *x* that minimizes the norm || *b - a x* ||.

Parameters

a : array_like, shape (M, N)

Input equation coefficients.

b : array_like, shape (M,) or (M, K)

Equation target values. If *b* is two-dimensional, the least squares solution is calculated for each of the *K* target sets.

rcond : float, optional

Cutoff for `small` singular values of *a*. Singular values smaller than *rcond* times the largest singular value are considered zero.

Returns

x : 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 Euclidian norm for each column in *b - a 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 : integer

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 returned as matrices.

Examples

Fit a line, `y = mx + c`, through some noisy data-points:

```
>>> 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 cuts the y-axis at more-or-less -1.

We can rewrite the line equation as `y = Ap`, where `A = [[x 1]]` and `p = [[m], [c]]`. Now use *lstsq* to solve for *p*:

```
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.]])
```

```
>>> m, c = np.linalg.lstsq(A, y)[0]
>>> print m, c
1.0 -0.95
```

Plot the data along with the fitted line:

```
>>> 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 inverse of a matrix.

Parameters

a : array_like, shape (M, M)

Matrix to be inverted

Returns

ainv : ndarray, shape (M, M)

Inverse of the matrix *a*

Raises

LinAlgError :

If *a* is singular or not square.

Examples

```
>>> a = np.array([[1., 2.], [3., 4.]])
>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.dot(a, np.linalg.inv(a))
array([[ 1.,  0.],
       [ 0.,  1.]])
```

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 (M, N)

Matrix to be pseudo-inverted.

rcond : float

Cutoff for *small* singular values. Singular values smaller than rcond*largest_singular_value are considered zero.

Returns

B : ndarray (N, M)

The pseudo-inverse of *a*. If *a* is an np.matrix instance, then so is *B*.

Raises

LinAlgError :

In case SVD computation does not converge.

Examples

```
>>> 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*)

Find the 'inverse' of a N-d array

The result is an inverse corresponding to the operation tensordot(a, b, ind), ie.,

x == tensordot(tensordot(tensorinv(a), a, ind), x, ind)
== tensordot(tensordot(a, tensorinv(a), ind), x, ind)

for all x (up to floating-point accuracy).

Parameters

a : array_like

Tensor to 'invert'. Its shape must 'square', ie., prod(a.shape[:ind]) == prod(a.shape[ind:])

ind : integer > 0

How many of the first indices are involved in the inverse sum.

Returns

b : array, shape a.shape[:ind]+a.shape[ind:]

Raises LinAlgError if a is singular or not square :

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`	# Error object

exception `LinAlgError`

3.8 Random sampling (`numpy.random`)

3.8.1 Simple random data

`rand`(d0,d1,...,dn)	Random values in a given shape.
`randn`(d0,d1,...,dn)	Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).
`randint`(low[,high,size])	Return random integers x such that low <= x < high.
`random_integers`(low[,high,size])	Return random integers x such that low <= x <= high.
`random_sample`([size])	Return random floats in the half-open interval [0.0, 1.0).
`bytes`(length)	Return random bytes.

`rand`(*d0, d1, ..., dn*)

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

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

randn (*d0, d1, ..., dn*)
Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).

Note: This is a convenience function. If you want an
interface that takes a tuple as the first argument use numpy.random.standard_normal(shape_tuple).

randint (*low, high=None, size=None*)
Return random integers x such that low <= x < high.

If high is None, then 0 <= x < low.

random_integers (*low, high=None, size=None*)
Return random integers x such that low <= x <= high.

If high is None, then 1 <= x <= low.

random_sample (*size=None*)
Return random floats in the half-open interval [0.0, 1.0).

bytes (*length*)
Return random bytes.

Parameters
length : int
Number of random bytes.

Returns
out : str
String of length *N*.

Examples

```
>>> np.random.bytes(10)
' eh\x85\x022SZ\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 (*x*)
Modify a sequence in-place by shuffling its contents.

permutation (*x*)
Randomly permute a sequence, or return a permuted range.

Parameters

x : int or array_like

If *x* is an integer, randomly permute `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

```
>>> 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

beta(a,b[,size])	The Beta distribution over `[0, 1]`.
binomial(n,p[,size])	Draw samples from a binomial distribution.
chisquare(df[,size])	Draw samples from a chi-square distribution.
mtrand.dirichlet(alpha[,size])	Draw samples from the Dirichlet distribution.
exponential([scale,size])	Exponential distribution.
f(dfnum,dfden[,size])	Draw samples from a F distribution.
gamma(shape[,scale,size])	Draw samples from a Gamma distribution.
geometric(p[,size])	Draw samples from the geometric distribution.
gumbel([loc,scale,size])	Gumbel distribution.
hypergeometric(ngood,nbad,nsample[,size])	Draw samples from a Hypergeometric distribution.
laplace([loc,scale,size])	Laplace or double exponential distribution.
logistic([loc,scale,size])	Draw samples from a Logistic distribution.
lognormal([mean,sigma,size])	Return samples drawn from a log-normal distribution.
logseries(p[,size])	Draw samples from a Logarithmic Series distribution.
multinomial(n,pvals[,size])	Draw samples from a multinomial distribution.
multivariate_normal(mean,cov[,size])	Draw random samples from a multivariate normal distribution.
negative_binomial(n,p[,size])	Negative Binomial distribution.
noncentral_chisquare(df,nonc[,size])	Draw samples from a noncentral chi-square distribution.
noncentral_f(dfnum,dfden,nonc[,size])	Noncentral F distribution.
normal([loc,scale,size])	Draw random samples from a normal (Gaussian) distribution.
pareto(a[,size])	Draw samples from a Pareto distribution with specified shape.
poisson([lam,size])	Poisson distribution.
power(a[,size])	Power distribution.
rayleigh([scale,size])	Rayleigh distribution.
standard_cauchy([size])	Standard Cauchy with mode=0.
standard_exponential([size])	Standard exponential distribution (scale=1).
standard_gamma(shape[,size])	Standard Gamma distribution.
standard_normal([size])	Standard Normal distribution (mean=0, stdev=1).
standard_t(df[,size])	Standard Student's t distribution with df degrees of freedom.

beta (*a, b, size=None*)

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 ouput is packed according to the size given.

Returns

out : ndarray

Array of the given shape, containing values drawn from a Beta distribution.

binomial (*n, p, size=None*)

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, >= 0 and <=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}

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 <=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

Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill, Fifth Edition, 2002.

Weisstein, Eric W. "Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

Examples

Draw samples from the distribution:

```
>>> 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.

```
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%.
```

chisquare(*df, size=None*)

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

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:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

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 Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

References

Wikipedia, "Chi-square distribution", http://en.wikipedia.org/wiki/Chi-square_distribution

Examples

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

dirichlet (*alpha, size=None*)

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

exponential(*scale=1.0, size=None*)

Exponential distribution.

Its probability density function is

$$f(x; \lambda) = \lambda \exp(-\lambda x),$$

for `x > 0` and 0 elsewhere. $lambda$ is known as the rate parameter.

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 [2], or the time between page requests to Wikipedia .

Parameters

scale : float

The rate parameter, λ.

size : tuple of ints

Number of samples to draw. The output is shaped according to *size*.

References

"Poisson Process", Wikipedia, http://en.wikipedia.org/wiki/Poisson_process

f(*dfnum, dfden, size=None*)

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 * n * 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.

[2] Peyton Z. Peebles Jr., "Probability, Random Variables and Random Signal Principles", 4th ed, 2001, p. 57.

References

Wikipedia, "F-distribution", http://en.wikipedia.org/wiki/F-distribution

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:

```
>>> 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 :

```
>>> 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(*shape, scale=1.0, size=None*)

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

Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * 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

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where k is the shape and θ the scale, and Γ 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

Wikipedia, "Gamma-distribution", http://en.wikipedia.org/wiki/Gamma-distribution

Examples

Draw samples from the distribution:

```
>>> 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:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*((exp(-bins/scale))/\
    (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

geometric(*p, size=None*)

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, ...`.

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:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

gumbel(*loc=0.0, scale=1.0, size=None*)

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

$$p(x) = \frac{e^{-(x - \mu)/ \beta}}{\beta} e^{ -e^{-(x - \mu)/ \beta}},$$

where μ is the mode, a location parameter, and β 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

Reiss, R.-D. and Thomas M. (2001), Statistical Analysis of Extreme Values, from Insurance, Finance, Hydrology and Other Fields, Birkhauser Verlag, Basel: Boston : Berlin.

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:

```
>>> 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:

```
>>> 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(*ngood, nbad, nsample, size=None*)

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

$$P(x) = \frac{\binom{m}{n}\binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \le x \le m$ and $n + m - N \le x \le n$

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

Weisstein, Eric W. "Hypergeometric Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypergeometricDistribution.html

Examples

Draw samples from the distribution:

```
>>> 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?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

laplace(*loc=0.0, scale=1.0, size=None*)

Laplace or double exponential distribution.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails.

Parameters

loc : float

The position, μ, of the distribution peak.

scale : float

λ, the exponential decay.

logistic(*loc=0.0, scale=1.0, size=None*)

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 μ = 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

Weisstein, Eric W. "Logistic Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogisticDistribution.html

Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
```

# plot against distribution

```
>>> def logist(x, loc, scale):
...     return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/\
... logist(bins, loc, scale).max())
>>> plt.show()
```

lognormal(*mean=0.0, sigma=1.0, size=None*)

Return samples drawn from a log-normal distribution.

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.

Parameters

mean : float
    Mean value of the underlying normal distribution

sigma : float, >0.
    Standard deviation of the underlying normal 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.lognorm
    probability density function, distribution, cumulative density function, etc.

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 μ is the mean and σ 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

Reiss, R.D., Thomas, M.(2001), Statistical Analysis of Extreme Values, Birkhauser Verlag, Basel, pp 31-32.

Examples

Draw samples from the distribution:

```
>>> 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:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> 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.

```
>>> # 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(*p, size=None*)

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

Fisher, R.A,, A.S. Corbet, and C.B. Williams. 1943. The relation between the number of species and the number of individuals in a random sample of an animal population. Journal of Animal Ecology, 12:42-58.

Wikipedia, "Logarithmic-distribution", http://en.wikipedia.org/wiki/Logarithmic-distribution

Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution

```
>>> 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(*n, pvals, size=None*)

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, ..., 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 `sum(pvals[:-1]) <= 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:

```
>>> 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:

```
>>> 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:

```
>>> np.random.multinomial(100, [1/7.]*5)
array([13, 16, 13, 16, 42])
```

multivariate_normal(*mean, cov, [size]*)

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 (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis
```

```
>>> 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

R.O. Duda, P.E. Hart, and D.G. Stork, “Pattern Classification,” 2nd ed., Wiley, 2001.

Examples

```
>>> 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:

```
>>> print list( (x[0,0,:] - mean) < 0.6 )
[True, True]
```

negative_binomial(*n, p, size=None*)
Negative Binomial distribution.

noncentral_chisquare(*df, nonc, size=None*)
Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

Parameters
df : int
Degrees of freedom.
nonc : float
Non-centrality.
size : tuple of ints
Shape of the output.

noncentral_f(*dfnum, dfden, nonc, size=None*)
Noncentral F distribution.

normal(*loc=0.0, scale=1.0, size=None*)
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 , is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions 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 .

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`
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 μ is the mean and σ the standard deviation. The square of the standard deviation, σ^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 $x - \sigma$). This implies that `numpy.random.normal` is more likely to return samples lying close to the mean, rather than those far away.

References

P. R. Peebles Jr., "Central Limit Theorem" in "Probability, Random Variables and Random Signal Principles", 4th ed., 2001, pp. 51, 51, 125.

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> 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:

```
>>> 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`(*a, size=None*)

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

Pareto, V. (1896). Course of Political Economy. Lausanne.

Wikipedia, "Pareto distribution", http://en.wikipedia.org/wiki/Pareto_distribution

Examples

Draw samples from the distribution:

```
>>> a, 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:

```
>>> 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`(*lam=1.0, size=None*)
Poisson distribution.

`power`(*a, size=None*)
Power distribution.

`rayleigh`(*scale=1.0, size=None*)
Rayleigh distribution.

`standard_cauchy`(*size=None*)
Standard Cauchy with mode=0.

`standard_exponential`(*size=None*)
Standard exponential distribution (scale=1).

`standard_gamma`(*shape, size=None*)
Standard Gamma distribution.

`standard_normal`(*size=None*)
Standard Normal distribution (mean=0, stdev=1).

`standard_t`(*df, size=None*)
Standard Student's t distribution with df degrees of freedom.

`triangular`(*left, mode, right, size=None*)
Triangular distribution starting at left, peaking at mode, and ending at right (left <= mode <= right).

`uniform`(*low=0.0, high=1.0, size=1*)
Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, 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:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
```

```
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> 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 (*mu=0.0, kappa=1.0, size=None*)

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 distribution 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 μ is the mode and κ 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

von Mises, Richard, 1964, Mathematical Theory of Probability and Statistics (New York: Academic Press).

Examples

Draw samples from the distribution:

```
>>> 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:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = arange(-pi, pi, 2*pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

`wald`(*mean, scale, size=None*)
Wald (inverse Gaussian) distribution.

`weibull`(*a, size=None*)
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 λ 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 * n * 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

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

References

Waloddi Weibull, 1951 "A Statistical Distribution Function of Wide Applicability", Journal Of Applied Mechanics ASME Paper.

Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> def weib(x,n,a):
...     return (a/n)*(x/n)**(a-1)*exp(-(x/n)**a)
```

```
>>> count, bins, ignored = plt.hist(numpy.random.weibull(5.,1000))
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> x = arange(1,100.)/50.
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

zipf(*a, size=None*)

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 ζ 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

Wikipedia, "Zeta distribution", http://en.wikipedia.org/wiki/Zeta_distribution

Zipf, George Kingsley (1932): Selected Studies of the Principle of Relative Frequency in Language. Cambridge (Mass.).

Examples

Draw samples from the distribution:

```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> 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 = 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

`mtrand.RandomState`([seed])	Container for the Mersenne Twister PRNG.
`seed`([seed])	Seed the generator.
`get_state`()	Return a tuple representing the internal state of the generator:
`set_state`(state)	Set the state from a tuple.

class RandomState ()

Container for the Mersenne Twister PRNG.

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 numpy array filled with generated values is returned. If size is a tuple, then a numpy array with that shape is filled and returned.

Parameters

seed : array_like, int, optional

Random seed initializing the PRNG. Can be an integer, an array (or other sequence) of integers of any length, or `None`. 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.

seed(*seed=None*)

Seed the generator.

seed can be an integer, an array (or other sequence) of integers of any length, or None. 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.

get_state()

Return a tuple representing the internal state of the generator:

```
('MT19937', int key[624], int pos, int has_gauss, float cached_gaussian)
```

set_state(*state*)

Set the state from a tuple.

state = ('MT19937', int key[624], int pos, int has_gauss, float cached_gaussian)

For backwards compatibility, the following form is also accepted although it is missing some information about the cached Gaussian value.

state = ('MT19937', int key[624], int pos)

3.9 Sorting and searching

3.9.1 Sorting

`sort`(a[,axis,kind,order])	Return a sorted copy of an array.
`lexsort`(keys[,axis])	Perform an indirect sort using a list 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:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```
>>> 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:

```
>>> 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')])
```

lexsort (*keys, axis=-1*)

Perform an indirect sort using a list of keys.

Imagine three input keys, `a`, `b` and `c`. These can be seen as columns in a spreadsheet. The first row of the spreadsheet would therefore be `a[0], b[0], c[0]`. Lexical sorting orders the different rows by first sorting on the on first column (key), then the second, and so forth. At each step, the previous ordering is preserved when equal keys are encountered.

Parameters

keys : (k,N) array or tuple containing k (N,)-shaped sequences

The *k* different "columns" to be sorted. The last column is the primary sort column.

axis : int, optional

Axis to be indirectly sorted. By default, sort over the last axis.

Returns

indices : (N,) ndarray of ints

Array of indices that sort the keys along the specified axis.

See Also:

argsort
Indirect sort.

ndarray.sort
In-place sort.

sort
Return a sorted copy of an array.

Examples

Sort names: first by surname, then by name.

```
>>> surnames =    ('Hertz',    'Galilei', 'Hertz')
>>> first_names = ('Heinrich', 'Galileo', 'Gustav')
>>> ind = np.lexsort((first_names, surnames))
>>> ind
array([1, 2, 0])
```

```
>>> [surnames[i] + ", " + first_names[i] for i in ind]
['Galilei, Galileo', 'Hertz, Gustav', 'Hertz, Heinrich']
```

Sort two columns of numbers:

```
>>> a = [1,5,1,4,3,4,4] # First column
>>> b = [9,4,0,4,0,2,1] # Second column
>>> ind = np.lexsort((b,a)) # Sort by second, then first column
>>> 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 the first elements are sorted. For each first element, the second elements are also sorted.

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:

```
>>> 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, optional

Axis along which to sort. If not given, 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.

Examples

One dimensional array:

```
>>> x = np.array([3, 1, 2])
>>> np.argsort(x)
array([1, 2, 0])
```

Two-dimensional array:

```
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])
```

```
>>> np.argsort(x, axis=0)
array([[0, 1],
       [1, 0]])
```

```
>>> np.argsort(x, axis=1)
array([[0, 1],
       [0, 1]])
```

Sorting with keys:

```
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

```
>>> np.argsort(x, order=('x','y'))
array([1, 0])
```

```
>>> np.argsort(x, order=('y','x'))
array([0, 1])
```

sort (*axis=-1, kind='quicksort', order=None*)

Sort an array, in-place.

Parameters

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

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

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

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([5 + 2j, 3 - 1j, 6 - 2j, 2 - 3j, 1 - 5j])
array([ 1.-5.j,  2.-3.j,  3.-1.j,  5.+2.j,  6.-2.j])
```

3.9.2 Searching

argmax(a[,axis])	Indices of the maximum values along an axis.
nanargmax(a[,axis])	Return indices of the maximum values over an axis, ignoring NaNs.
argmin(a[,axis])	Return the indices of the minimum values along an axis.
nanargmin(a[,axis])	Return indices of the minimum values along an axis, ignoring NaNs.
argwhere(a)	Find the indices of array elements that are non-zero, grouped by element.
nonzero(a)	Return the indices of the elements that are non-zero.
flatnonzero(a)	Return indices that are non-zero in the flattened version of a.
where(condition,[x,y])	Return elements, either from *x* or *y*, depending on *condition*.
searchsorted(a,v[,side])	Find indices where elements should be inserted to maintain order.
extract(condition,arr)	Return the elements of an array that satisfy some condition.

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, int

Array of indices into the array. It has the same shape as *a*, except with *axis* removed.

See Also:

argmin
Indices of the minimum values along an axis.

amax
The maximum value along a given axis.

unravel_index
Convert a flat index into an index tuple.

Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

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

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmax(a)
0
>>> np.nanargmax(a)
1
>>> np.nanargmax(a, axis=0)
array([1, 1])
>>> np.nanargmax(a, axis=1)
array([1, 0])
```

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 along an axis, ignoring NaNs.

See Also:

nanargmax
corresponding function for maxima; see for details.

argwhere(*a*)
Find the indices of array elements that are non-zero, grouped by element.

Parameters
a : array_like
Input data.
Returns
index_array : ndarray
Indices of elements that are non-zero. Indices are grouped by element.

See Also:

where, nonzero

Notes

`np.argwhere(a)` is the same as `np.transpose(np.nonzero(a))`.

The output of `argwhere` is not suitable for indexing arrays. For this purpose use `where(a)` instead.

Examples

```
>>> x = np.arange(6).reshape(2,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argwhere(x>1)
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
```

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

```
>>> x = np.eye(3)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))
```

```
>>> x[np.nonzero(x)]
array([ 1.,  1.,  1.])
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])
```

flatnonzero(*a*)

Return indices that are non-zero in the flattened version of a.

This is equivalent to a.ravel().nonzero()[0].

Parameters

a : ndarray

Input array.

Returns

res : ndarray

Output array, containing the indices of the elements of *a.ravel()* that are non-zero.

See Also:

nonzero
Return the indices of the non-zero elements of the input array.

ravel
Return a 1-D array containing the elements of the input array.

Examples

```
>>> x = np.arange(-2, 3)
>>> x
array([-2, -1,  0,  1,  2])
>>> np.flatnonzero(x)
array([0, 1, 3, 4])
```

```
>>> x.ravel()[np.flatnonzero(x)]
array([-2, -1,  1,  2])
```

where(*condition, [x, y]*)

Return elements, either from *x* or *y*, depending on *condition*.

If only *condition* is given, return `condition.nonzero()`.

Parameters

condition : array_like, bool

When True, yield *x*, otherwise yield *y*.

x, y : array_like, optional

Values from which to choose.

Returns

out : ndarray or tuple of ndarrays

If both *x* and *y* are specified, the output array, shaped like *condition*, 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

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

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

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 of shape (N,)

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
In-place sort.

histogram
Produce histogram from 1-D data.

Notes

Binary search is used to find the required insertion points.

Examples

```
>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])
```

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

Input array of the same size as *condition*.

See Also:

`take`, `put`, `putmask`

Examples

```
>>> arr = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> condition = np.mod(arr, 3)==0
>>> condition
array([[False, False,  True, False],
       [False,  True, False, False],
       [ True, False, False,  True]], dtype=bool)
>>> np.extract(condition, arr)
array([ 3,  6,  9, 12])
```

If *condition* is boolean:

```
>>> arr[condition]
array([ 3,  6,  9, 12])
```

3.10 Logic functions

3.10.1 Truth value testing

`all`(a[,axis,out])	Returns True if all elements evaluate to True.
`any`(a[,axis,out])	Test whether any elements of an array evaluate to True along an axis.

all (*a, axis=None, out=None*)

Returns True if all elements evaluate to True.

Parameters

a : array_like

Input array.

axis : int, optional

Axis over which to perform the operation. If None, use a flattened input array and return a bool.

out : ndarray, optional

Array into which the result is placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

out : ndarray, bool

A logical AND is performed along *axis*, and the result placed in *out*. If *out* was not specified, a new output array is created.

See Also:

`ndarray.all`
equivalent method

Notes

Since NaN is not equal to zero, NaN evaluates to True.

Examples

```
>>> np.all([[True,False],[True,True]])
False
```

```
>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False], dtype=bool)
```

```
>>> np.all([-1, 4, 5])
True
```

```
>>> np.all([1.0, np.nan])
True
```

`any`(*a, axis=None, out=None*)

Test whether any elements of an array evaluate to True along an axis.

Parameters

a : array_like

Input array.

axis : int, optional

Axis over which to perform the operation. If None, use a flattened input array and return a bool.

out : ndarray, optional

Array into which the result is placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

out : ndarray

A logical OR is performed along *axis*, and the result placed in *out*. If *out* was not specified, a new output array is created.

See Also:

ndarray.any
equivalent method

Notes

Since NaN is not equal to zero, NaN evaluates to True.

Examples

```
>>> np.any([[True, False], [True, True]])
True
```

```
>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False], dtype=bool)
```

```
>>> np.any([-1, 0, 5])
True
```

```
>>> np.any(np.nan)
True
```

3.10.2 Array contents

isfinite(x[,out])	Returns True for each element that is a finite number.
isinf(x[,out])	Shows which elements of the input are positive or negative infinity. Returns a numpy boolean scalar or array resulting from an element-wise test for positive or negative infinity.
isnan(x[,out])	Returns a numpy boolean scalar or array resulting from an element-wise test for Not a Number (NaN).
isneginf(x[,y])	Return True where x is -infinity, and False otherwise.
isposinf(x[,y])	Shows which elements of the input are positive infinity.

isfinite(*x, [out]*)
Returns True for each element that is a finite number.

Shows which elements of the input are finite (not infinity or not Not a Number).

Parameters
x : array_like
Input values.

y : array_like, optional
A boolean array with the same shape and type as *x* to store the result.

Returns
y : ndarray, bool
For scalar input data, the result is a new numpy boolean with value True if the input data is finite; otherwise the value is False (input is either positive infinity, negative infinity or Not a Number).

For array input data, the result is an numpy 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). If the second argument is supplied then an numpy integer array is returned with values 0 or 1 corresponding to False and True, respectively.

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).

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 second argument is also supplied with scalar input or if first and second arguments have different shapes.

Examples

```
>>> np.isfinite(1)
True
>>> np.isfinite(0)
True
>>> np.isfinite(np.nan)
False
>>> np.isfinite(np.inf)
False
>>> np.isfinite(np.NINF)
False
>>> np.isfinite([np.log(-1.),1.,np.log(0)])
array([False,  True, False], dtype=bool)
>>> x=np.array([-np.inf, 0., np.inf])
>>> y=np.array([2,2,2])
>>> np.isfinite(x,y)
array([0, 1, 0])
>>> y
array([0, 1, 0])
```

isinf (*x, [out]*)
Shows which elements of the input are positive or negative infinity. Returns a numpy boolean scalar or array resulting from an element-wise test for positive or negative infinity.

Parameters
x : array_like
input values
y : array_like, optional

An array with the same shape as *x* to store the result.

Returns

y : {ndarray, bool}

For scalar input data, the result is a new numpy boolean with value True if the input data is positive or negative infinity; otherwise the value is False.

For array input data, the result is an numpy 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 the second argument is supplied then an numpy integer array is returned with values 0 or 1 corresponding to False and True, respectively.

See Also:

`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 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. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity.

Errors result if second argument is also supplied with scalar input or if first and second arguments have different shapes.

Examples

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

`isnan` (*x, [out]*)

Returns a numpy boolean scalar or array resulting from an element-wise test for Not a Number (NaN).

Parameters

x : array_like

input data.

Returns

y : {ndarray, bool}

For scalar input data, the result is a new numpy boolean with value True if the input data is NaN; otherwise the value is False.
For array input data, the result is an numpy boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is Not a Number; otherwise the values are False.

See Also:

isinf
Tests for infinity.

isneginf
Tests for negative infinity.

isposinf
Tests for positive infinity.

isfinite
Shows which elements are not: Not a number, positive infinity 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.

Examples

```
>>> 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*)
Return True where x is -infinity, and False otherwise.

Parameters
x : array_like
The input array.
y : array_like
A boolean array with the same shape as *x* to store the result.

Returns
y : ndarray
A boolean array where y[i] = True only if x[i] = -Inf.

See Also:

isposinf, isfinite

Examples

```
>>> np.isneginf([-np.inf, 0., np.inf])
array([ True, False, False], dtype=bool)
```

isposinf(*x, y=None*)
Shows which elements of the input are positive infinity.

Returns a numpy array resulting from an element-wise test for positive infinity.

Parameters

x : array_like

The input array.

y : array_like

A boolean array with the same shape as *x* to store the result.

Returns

y : ndarray

A numpy 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 positive infinity and values False where the element of the input is not positive infinity.

If second argument is supplied then an numpy integer array is returned with values 1 where the corresponding element of the input is positive positive infinity.

See Also:

isinf
Shows which elements are negative or positive infinity.

isneginf
Shows which elements are negative infinity.

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. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity.

Errors result if second argument is also supplied with scalar input or if first and second arguments have different shapes.

Examples

```
>>> np.isposinf(np.PINF)
array(True, dtype=bool)
>>> np.isposinf(np.inf)
array(True, dtype=bool)
>>> np.isposinf(np.NINF)
array(False, dtype=bool)
>>> np.isposinf([-np.inf, 0., np.inf])
array([False, False,  True], dtype=bool)
>>> x=np.array([-np.inf, 0., np.inf])
>>> y=np.array([2,2,2])
>>> np.isposinf(x,y)
array([1, 0, 0])
>>> y
array([1, 0, 0])
```

3.10.3 Array type testing

iscomplex(x)	Return a bool array, True if element is complex (non-zero imaginary part).
iscomplexobj(x)	Return True if x is a complex type or an array of complex numbers.
isfortran(a)	Returns True if array is arranged in Fortran-order and dimension > 1.
isreal(x)	Returns a bool array where True if the corresponding input element is real.
isrealobj(x)	Return True if x is not a complex type.
isscalar(num)	Returns True if the type of num is a scalar type.

iscomplex(*x*)

Return a bool array, True if element is complex (non-zero imaginary part).

For scalars, return a boolean.

Parameters

x : array_like

Input array.

Returns

out : ndarray, bool

Output array.

Examples

```
>>> x = np.array([1,2,3.j])
>>> np.iscomplex(x)
array([False, False,  True], dtype=bool)
```

iscomplexobj(*x*)

Return True if x is a complex type or an array of complex numbers.

Unlike iscomplex(x), complex(3.0) is considered a complex object.

isfortran(*a*)

Returns True if array is arranged in Fortran-order and dimension > 1.

Parameters

a : ndarray

Input array to test.

isreal(*x*)

Returns a bool array where True if the corresponding input element is real.

True if complex part is zero.

Parameters

x : array_like

Input array.

Returns

out : ndarray, bool

Boolean array of same shape as *x*.

Examples

```
>>> 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 not a complex type.

Unlike isreal(x), complex(3.0) is considered a complex object.

`isscalar`(*num*)

Returns True if the type of num is a scalar type.

Parameters

num : any

Input argument.

Returns

val : bool

True if *num* is a scalar type, False if it is not.

Examples

```
>>> np.isscalar(3.1)
True
>>> np.isscalar([3.1])
False
>>> np.isscalar(False)
True
```

3.10.4 Logical operations

`logical_and`(x1,x2[,out])	Compute the truth value of x1 AND x2 elementwise.
`logical_or`(x1,x2[,out])	Compute the truth value of x1 OR x2 elementwise.
`logical_not`(x[,out])	Compute the truth value of NOT x elementwise.
`logical_xor`(x1,x2[,out])	Compute the truth value of x1 XOR x2 elementwise.

`logical_and`(*x1, x2, [out]*)

Compute the truth value of x1 AND x2 elementwise.

Parameters

x1, x2 : array_like

Logical AND is applied to the elements of *x1* and *x2*. They have to 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 elements of *x1* and *x2*.

See Also:

`logical_or`, `logical_not`, `logical_xor`, `bitwise_and`

Examples

```
>>> 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 (*x1, x2, [out]*)
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 : {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

```
>>> np.logical_or(True, False)
True
>>> np.logical_or([True, False], [False, False])
array([ True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_or(x < 1, x > 3)
array([ True, False, False, False,  True], dtype=bool)
```

logical_not (*x, [out]*)
Compute the truth value of NOT x elementwise.

Parameters
x : array_like
Logical NOT is applied to the elements of *x*.

Returns
y : {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

```
>>> np.logical_not(3)
False
>>> np.logical_not([True, False, 0, 1])
array([False,  True,  True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_not(x<3)
array([False, False, False,  True,  True], dtype=bool)
```

logical_xor (*x1, x2, [out]*)

Compute the truth value of x1 XOR x2 elementwise.

Parameters

x1, x2 : array_like

Logical XOR is applied to the elements of *x1* and *x2*. They have to be of the same shape.

Returns

y : {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:

logical_and, logical_or, logical_not, bitwise_xor

Examples

```
>>> np.logical_xor(True, False)
True
>>> np.logical_xor([True, True, False, False], [True, False, True, False])
array([False,  True,  True, False], dtype=bool)
```

```
>>> x = np.arange(5)
>>> np.logical_xor(x < 1, x > 3)
array([ True, False, False, False,  True], dtype=bool)
```

3.10.5 Comparison

allclose(a,b[,rtol,atol])	Returns True if two arrays are element-wise equal within a tolerance.
array_equal(a1,a2)	True if two arrays have the same shape and elements, False otherwise.
array_equiv(a1,a2)	Returns True if input arrays are shape consistent and all elements equal.

allclose (*a, b, rtol=1.0000000000000001e-05, 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 (*rtol* * *b*) and the absolute difference (*atol*) are added together to compare against the absolute difference between *a* and *b*.

Parameters

a, b : array_like

Input arrays to compare.

rtol : Relative tolerance

The relative difference is equal to *rtol* * *b*.

atol : Absolute tolerance

The absolute difference is equal to *atol*.

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, alltrue, sometrue

Notes

If the following equation is element-wise True, then allclose returns True.

absolute(*a* - *b*) <= (*atol* + *rtol* * absolute(*b*))

Examples

```
>>> np.allclose([1e10,1e-7], [1.00001e10,1e-8])
False
>>> np.allclose([1e10,1e-8], [1.00001e10,1e-9])
True
>>> np.allclose([1e10,1e-8], [1.0001e10,1e-9])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan])
False
```

array_equal(*a1, a2*)

True if two arrays have the same shape and elements, False otherwise.

Parameters

a1 : array_like

First input array.

a2 : array_like

Second input array.

Returns

b : {True, False}

Returns True if the arrays are equal.

Examples

```
>>> np.array_equal([1,2],[1,2])
True
>>> np.array_equal(np.array([1,2]),np.array([1,2]))
True
>>> np.array_equal([1,2],[1,2,3])
False
>>> np.array_equal([1,2],[1,4])
False
```

array_equiv(*a1, a2*)

Returns True if input arrays are shape consistent and all elements equal.

Parameters

a1 : array_like

Input array.

a2 : array_like

Input array.

Returns

out : bool

True if equivalent, False otherwise.

Examples

```
>>> np.array_equiv([1,2],[1,2])
>>> True
>>> np.array_equiv([1,2],[1,3])
>>> False
>>> np.array_equiv([1,2], [[1,2],[1,2]])
>>> True
>>> np.array_equiv([1,2], [[1,2],[1,3]])
>>> False
```

greater(x1,x2[,out])	Return (x1 > x2) element-wise.
greater_equal(x1,x2[,out])	Element-wise True if first array is greater or equal than second array.
less(x1,x2[,out])	Returns (x1 < x2) element-wise.
less_equal(x1,x2[,out])	Returns (x1 <= x2) element-wise.
equal(x1,x2[,out])	Returns elementwise x1 == x2 in a bool array.
not_equal(x1,x2[,out])	Return (x1 != x2) element-wise.

greater (*x1, x2, [out]*)

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:

greater_equal, less, less_equal, equal, not_equal

Examples

```
>>> np.greater([4,2],[2,2])
array([ True, False], dtype=bool)
```

If the inputs are ndarrays, then np.greater is equivalent to '>'.

```
>>> a = np.array([4,2])
>>> b = np.array([2,2])
>>> a > b
array([ True, False], dtype=bool)
```

greater_equal (*x1, x2, [out]*)

Element-wise True if first array is greater or equal than second array.

Parameters

x1, x2 : array_like

Input arrays.

Returns

out : ndarray, bool

Output array.

See Also:

greater, less, less_equal, equal

Examples

```
>>> np.greater_equal([4,2],[2,2])
array([ True,  True], dtype=bool)
```

less (*x1, x2, [out]*)

Returns (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_equal

Examples

```
>>> np.less([1,2],[2,2])
array([ True, False], dtype=bool)
```

less_equal (*x1, x2, [out]*)

Returns (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

Examples

```
>>> np.less_equal([1,2,3],[2,2,2])
array([ True,  True, False], dtype=bool)
```

equal (*x1, x2, [out]*)

Returns elementwise x1 == x2 in a bool array.

Parameters

x1, x2 : array_like

Input arrays of the same shape.

Returns

out : boolean

The elementwise test *x1* == *x2*.

not_equal(*x1, x2, [out]*)

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.

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

```
>>> np.not_equal([1.,2.], [1., 3.])
array([False,  True], dtype=bool)
```

3.11 Binary operations

3.11.1 Elementwise bit operations

`bitwise_and`(x1,x2[,out])	Compute bit-wise AND of two arrays, element-wise.
`bitwise_or`(x1,x2[,out])	Compute bit-wise OR of two arrays, element-wise.
`bitwise_xor`(x1,x2[,out])	Compute bit-wise XOR of two arrays, element-wise.
`invert`(x[,out])	Compute bit-wise inversion, or bit-wise NOT, element-wise.
`left_shift`(x1,x2[,out])	Shift the bits of an integer to the left.
`right_shift`(x1,x2[,out])	Shift the bits of an integer to the right.

bitwise_and(*x1, x2, [out]*)

Compute bit-wise AND of two arrays, element-wise.

When calculating the bit-wise AND between two elements, `x` and `y`, each element is first converted to its binary representation (which works just like the decimal system, only now we're using 2 instead of 10):

$$x = \sum_{i=0}^{W-1} a_i \cdot 2^i$$
$$y = \sum_{i=0}^{W-1} b_i \cdot 2^i,$$

where `W` is the bit-width of the type (i.e., 8 for a byte or uint8), and each a_i and b_j is either 0 or 1. For example, 13 is represented as `00001101`, which translates to $2^4 + 2^3 + 2$.

The bit-wise operator is the result of

$$z = \sum_{i=0}^{i=W-1} (a_i \wedge b_i) \cdot 2^i,$$

where $\wedge$ is the AND operator, which yields one whenever both a_i and b_i are 1.

Parameters
- **x1, x2** : array_like
  Only integer types are handled (including booleans).

Returns
- **out** : array_like
  Result.

See Also:

`bitwise_or`, `bitwise_xor`, `logical_and`

`binary_repr`
Return the binary representation of the input number as a string.

Examples

We have seen that 13 is represented by `00001101`. Similary, 17 is represented by `00010001`. The bit-wise AND of 13 and 17 is therefore `000000001`, or 1:

```
>>> np.bitwise_and(13, 17)
1
```

```
>>> np.bitwise_and(14, 13)
12
>>> np.binary_repr(12)
'1100'
>>> 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`(*x1, x2, [out]*)

Compute bit-wise OR of two arrays, element-wise.

When calculating the bit-wise OR between two elements, `x` and `y`, each element is first converted to its binary representation (which works just like the decimal system, only now we are using 2 instead of 10):

$$x = \sum_{i=0}^{W-1} a_i \cdot 2^i$$

$$y = \sum_{i=0}^{W-1} b_i \cdot 2^i,$$

where `W` is the bit-width of the type (i.e., 8 for a byte or uint8), and each a_i and b_j is either 0 or 1. For example, 13 is represented as `00001101`, which translates to $2^4 + 2^3 + 2$.

The bit-wise operator is the result of

$$z = \sum_{i=0}^{i=W-1} (a_i \vee b_i) \cdot 2^i,$$

where $\vee$ is the OR operator, which yields one whenever either a_i or b_i is 1.

Parameters

x1, x2 : array_like

Only integer types are handled (including booleans).

Returns

out : array_like

Result.

See Also:

`bitwise_and`, `bitwise_xor`, `logical_or`

`binary_repr`
Return the binary representation of the input number as a string.

Examples

We've seen that 13 is represented by `00001101`. Similary, 16 is represented by `00010000`. The bit-wise OR of 13 and 16 is therefore `000111011`, or 29:

```
>>> np.bitwise_or(13, 16)
29
>>> np.binary_repr(29)
'11101'
```

```
>>> 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])
```

```
>>> np.bitwise_or(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])
>>> np.bitwise_or([True, True], [False, True])
array([ True,  True], dtype=bool)
```

`bitwise_xor` (*x1, x2, [out]*)
Compute bit-wise XOR of two arrays, element-wise.

When calculating the bit-wise XOR between two elements, `x` and `y`, each element is first converted to its binary representation (which works just like the decimal system, only now we are using 2 instead of 10):

$$x = \sum_{i=0}^{W-1} a_i \cdot 2^i$$
$$y = \sum_{i=0}^{W-1} b_i \cdot 2^i,$$

where `W` is the bit-width of the type (i.e., 8 for a byte or uint8), and each a_i and b_j is either 0 or 1. For example, 13 is represented as `00001101`, which translates to $2^4 + 2^3 + 2$.

The bit-wise operator is the result of

$$z = \sum_{i=0}^{i=W-1} (a_i \oplus b_i) \cdot 2^i,$$

where $\oplus$ is the XOR operator, which yields one whenever either a_i or b_i is 1, but not both.

Parameters
  x1, x2 : array_like
    Only integer types are handled (including booleans).

Returns
  out : ndarray
    Result.

See Also:

`bitwise_and`, `bitwise_or`, `logical_xor`

`binary_repr`
  Return the binary representation of the input number as a string.

Examples

We've seen that 13 is represented by `00001101`. Similary, 17 is represented by `00010001`. The bit-wise 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 (*x, [out]*)

Compute bit-wise inversion, or bit-wise NOT, element-wise.

When calculating the bit-wise NOT of an element `x`, each element is first converted to its binary representation (which works just like the decimal system, only now we're using 2 instead of 10):

$$x = \sum_{i=0}^{W-1} a_i \cdot 2^i$$

where `W` is the bit-width of the type (i.e., 8 for a byte or uint8), and each a_i is either 0 or 1. For example, 13 is represented as `00001101`, which translates to $2^4 + 2^3 + 2$.

The bit-wise operator is the result of

$$z = \sum_{i=0}^{i=W-1} (\neg a_i) \cdot 2^i,$$

where $\neg$ is the NOT operator, which yields 1 whenever a_i is 0 and yields 0 whenever a_i is 1.

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 [3]. A N-bit two's-complement system can represent every integer in the range -2^{N-1} to $+2^{N-1} - 1$.

Parameters

x1 : ndarray

Only integer types are handled (including booleans).

Returns

out : ndarray

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*:

```
>>> np.bitwise_not is np.invert
True
```

References

Examples

We've seen that 13 is represented by `00001101`. The invert or bit-wise NOT of 13 is then:

```
>>> 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'
```

[3] Wikipedia, "Two's complement", http://en.wikipedia.org/wiki/Two's_complement

The result depends on the bit-width:

```
>>> 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:

```
>>> np.invert(np.array([13], dtype=int8))
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8)
'11110010'
```

Booleans are accepted as well:

```
>>> np.invert(array([True, False]))
array([False,  True], dtype=bool)
```

left_shift (*x1, x2, [out]*)

Shift the bits of an integer to the left.

Bits are shifted to the left by appending *x2* 0s at the right of *x1*. Since the internal representation of numbers is in binary format, this operation is equivalent to multiplying *x1* by `2**x2`.

Parameters

x1 : array_like of integer type

Input values.

x2 : array_like of integer type

Number of zeros to append to *x1*.

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.left_shift(5, [1,2,3])
array([10, 20, 40])
```

right_shift (*x1, x2, [out]*)

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.right_shift(10, [1,2,3])
array([5, 2, 1])
```

3.11.2 Bit packing

packbits(myarray[,axis])	myarray : an integer type array whose elements should be packed to bits
unpackbits(myarray[,axis])	myarray - array of uint8 type where each element represents a bit-field that should be unpacked into a boolean output array

packbits (*myarray, axis=None*)

myarray : an integer type array whose elements should be packed to bits

This routine packs the elements of a binary-valued dataset into a NumPy array of type uint8 ('B') whose bits correspond to the logical (0 or nonzero) value of the input elements. The dimension over-which bit-packing is done is given by axis. The shape of the output has the same number of dimensions as the input (unless axis is None, in which case the output is 1-d).

Example: >>> a = array([[[1,0,1], ... [0,1,0]], ... [[1,1,0], ... [0,0,1]]]) >>> b = numpy.packbits(a,axis=-1) >>> b array([[[160],[64]],[[192],[32]]], dtype=uint8)

Note that 160 = 128 + 32
192 = 128 + 64

unpackbits (*myarray, axis=None*)

myarray - array of uint8 type where each element represents a bit-field
that should be unpacked into a boolean 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.

3.11.3 Output formatting

binary_repr(num[,width])	Return the binary representation of the input number as a string.

binary_repr(*num, width=None*)

Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two's complement of the number is returned, with respect to that width.

In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers [4]. 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`

Notes

binary_repr is equivalent to using *base_repr* with base 2, but about 25x faster.

References

Examples

```
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

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

```
>>> np.binary_repr(-3, width=4)
'1101'
```

[4] Wikipedia, "Two's complement", http://en.wikipedia.org/wiki/Two's_complement

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.

amin(*a, axis=None, out=None*)

Return the 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.

Returns

amin : ndarray

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

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

```
>>> 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])
```

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.

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

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

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a, axis=0)
array([2, 3])
>>> np.amax(a, axis=1)
array([1, 3])
```

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

```
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmax(a)
3.0
>>> np.nanmax(a, axis=0)
array([ 3.,  2.])
>>> np.nanmax(a, axis=1)
array([ 2.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> np.nanmax([1, 2, np.nan, np.NINF])
2.0
>>> np.nanmax([1, 2, np.nan, np.inf])
inf
```

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, 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

```
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmin(a)
1.0
>>> np.nanmin(a, axis=0)
array([ 1.,  2.])
>>> np.nanmin(a, axis=1)
array([ 1.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> np.nanmin([1, 2, np.nan, np.inf])
1.0
>>> np.nanmin([1, 2, np.nan, np.NINF])
-inf
```

ptp(*a, axis=None, out=None*)
Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

Parameters
a : array_like
Input values.

axis : int, optional
Axis along which to find the peaks. By default, flatten the array.

out : array_like
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type of the output values will be cast if necessary.

Returns
ptp : ndarray
A new array holding the result, unless *out* was specified, in which case a reference to *out* is returned.

Examples

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])
```

```
>>> np.ptp(x, axis=0)
array([2, 2])

>>> np.ptp(x, axis=1)
array([1, 1])
```

3.12.2 Averages and variances

average(a[,axis,weights,returned])	Return the weighted average of array over the specified axis.
mean(a[,axis,dtype,out])	Compute the arithmetic mean along the specified axis.
median(a[,axis,out,overwrite_input])	Compute the median along the specified axis.
std(a[,axis,dtype,out,...])	Compute the standard deviation along the specified axis.
var(a[,axis,dtype,out,...])	Compute the variance along the specified axis.

average (*a, axis=None, weights=None, returned=False*)
Return the weighted average of array over the specified axis.

Parameters

a : array_like
Data to be averaged.

axis : int, optional
Axis along which to average *a*. If *None*, averaging is done over the entire array irrespective of its shape.

weights : array_like, optional
The importance that each datum 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
Default is *False*. If *True*, the tuple (*average*, *sum_of_weights*) is returned, otherwise only the average is returned. Note that 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:

ma.average
average for masked arrays

Examples

```
>>> data = range(1,5)
>>> data
[1, 2, 3, 4]
>>> np.average(data)
2.5
>>> np.average(range(1,11), weights=range(10,0,-1))
4.0
```

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.

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:

average
Weighted average

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Examples

```
>>> a = np.array([[1,2],[3,4]])
>>> np.mean(a)
2.5
>>> np.mean(a,0)
array([ 2.,  3.])
>>> np.mean(a,1)
array([ 1.5,  3.5])
```

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 : {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_sorted` - i.e., `V_sorted[(N-1)/2]`, when N is odd. When N is even, it is the average of the two middle values of `V_sorted`.

Examples

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.median(a)
3.5
>>> np.median(a, axis=0)
array([ 6.5,  4.5,  2.5])
>>> np.median(a, axis=1)
array([ 7.,  2.])
>>> m = np.median(a, axis=0)
>>> out = np.zeros_like(m)
>>> np.median(a, axis=0, out=m)
array([ 6.5,  4.5,  2.5])
>>> m
array([ 6.5,  4.5,  2.5])
>>> b = a.copy()
>>> np.median(b, axis=1, overwrite_input=True)
```

```
array([ 7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.median(b, axis=None, overwrite_input=True)
3.5
>>> assert not np.all(a==b)
```

std(*a, 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 (biased estimate).

Returns

standard_deviation : {ndarray, scalar}; 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:

numpy.var
Variance

numpy.mean
Average

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `var = sqrt(mean(abs(x - x.mean())**2))`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, 0)
array([ 1.,  1.])
>>> np.std(a, 1)
array([ 0.5,  0.5])
```

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

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of 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.

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])
```

3.12.3 Correlating

corrcoef(x[,y,rowvar,bias])	Return correlation coefficients.
correlate(a,v[,mode])	Discrete, linear correlation of two 1-dimensional sequences.
cov(m[,y,rowvar,bias])	Estimate a covariance matrix, given data.

corrcoef (*x, y=None, rowvar=1, bias=0*)

Return correlation coefficients.

Please refer to the documentation for *cov* for more detail. The relationship between the correlation coefficient matrix, P, and the covariance matrix, C, is

$$P_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

The values of P are between -1 and 1.

See Also:

cov
Covariance matrix

correlate (*a, v, mode='valid'*)

Discrete, linear correlation of two 1-dimensional sequences.

This function is equivalent to

```
>>> np.convolve(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*.

See Also:

convolve
Discrete, linear convolution of two one-dimensional sequences.

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, $X = [x_1, x_2, ...x_N]^T$, 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:

```
>>> 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:

```
>>> 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:

```
>>> 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

`histogram`(a[,bins,range,normed,...])	Compute the histogram of a set of data.
`histogram2d`(x,y[,bins,range,normed,...])	Compute the bidimensional histogram of two data samples.
`histogramdd`(sample[,bins,range,normed,...])	Compute the multidimensional histogram of some data.
`bincount`(x,weights])	Return the number of occurrences of each value in x.
`digitize`(x,bins)	Return the index of the bin to which each value of x belongs.

histogram(*a, bins=10, range=None, normed=False, weights=None, new=None*)

Compute the histogram of a set of data.

Parameters

a : array_like

Input data.

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.min(), a.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 often not be equal to 1; 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 1.4.

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 `(length(hist)+1)`. With `new=False`, return the left bin edges (`length(hist)`).

See Also:

`histogramdd`

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

```
>>> np.histogram([1,2,1], bins=[0,1,2,3])
(array([0, 2, 1]), array([0, 1, 2, 3]))
```

histogram2d(*x, y, bins=10, range=None, normed=False, weights=None*)

Compute the bidimensional 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:

- the number of bins for the two dimensions (nx=ny=bins),
- the number of bins in each dimension (nx, ny = bins),
- the bin edges for the two dimensions (x_edges=y_edges=bins),
- 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 : 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 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 bidimensional 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 $\Delta x_i \Delta y_i$ 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 *histogrammdd*.

Examples

```
>>> 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,))
```

histogramdd(*sample, bins=10, range=None, normed=False, weights=None*)
Compute the multidimensional histogram of some data.

Parameters

sample : array_like

Data to histogram passed as a sequence of D arrays of length N, or as an (N,D) array.

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 explicitely 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_i* weighing each sample *(x_i, y_i, z_i, ...)*. Weights are normalized to 1 if normed is True. If normed is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

Returns

H : ndarray

The multidimensional histogram of sample x. See normed and weights for the different possible semantics.

edges : list

A list of D arrays describing the bin edges for each dimension.

See Also:

histogram
1D histogram

histogram2d
2D histogram

Examples

```
>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5,8,4), 6, 9, 5)
```

bincount (*x, weights=None*)

Return the number of occurrences of each value in x.

x must be a list of non-negative integers. The output, b[i], represents the number of times that i is found in x. If weights is specified, every occurrence of i at a position p contributes weights[p] instead of 1.

See also: histogram, digitize, unique.

digitize (*x, bins*)

Return the index of the bin to which each value of x 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.

Beyond the bounds of the bins 0 or len(bins) is returned as appropriate.

3.13 Mathematical functions

3.13.1 Trigonometric functions

sin(x[,out])	Trigonometric sine, element-wise.
cos(x[,out])	Cosine elementwise.
tan(x[,out])	Compute tangent element-wise.
arcsin(x[,out])	Inverse sine elementwise.
arccos(x[,out])	Trigonometric inverse cosine, element-wise.
arctan(x[,out])	Trigonometric inverse tangent, element-wise.
hypot(x1,x2[,out])	Given two sides of a right triangle, return its hypotenuse.
arctan2(x1,x2[,out])	Elementwise arc tangent of `x1/x2` choosing the quadrant correctly.
degrees(x[,out])	Convert angles from radians to degrees.
radians(x[,out])	Convert angles from degrees to radians.
unwrap(p[,discont,axis])	Unwrap by changing deltas between values to 2*pi complement.

sin(*x, [out]*)

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 $x = 3\pi/2$ to +1 for $\pi/2$. 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:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> 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:

```
>>> 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()
```

`cos` (*x, [out]*)
Cosine elementwise.

Parameters
x : array_like
Input array in radians.

Returns
out : ndarray
Output array of same shape as *x*.

Examples

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
```

`tan` (*x, [out]*)
Compute tangent element-wise.

Parameters
x : array_like
Angles in radians.

Returns
y : ndarray
The corresponding tangent values.

Examples

```
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
```

`arcsin` (*x, [out]*)
Inverse sine elementwise.

Parameters
x : array_like
y-coordinate on the unit circle.

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`, `arctan`, `arctan2`

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

Wikipedia, "Inverse trigonometric function", http://en.wikipedia.org/wiki/Arcsin

Examples

```
>>> np.arcsin(1)      # pi/2
1.5707963267948966
>>> np.arcsin(-1)     # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

`arccos` (*x, [out]*)

Trigonometric inverse cosine, element-wise.

The inverse of *cos* so that, if `y = cos(x)`, then `x = arccos(y)`.

Parameters

x : array_like

x-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

Returns

angle : ndarray

The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians [0, pi]. If *x* is a scalar then a scalar is returned, otherwise an array of the same shape as *x* is returned.

See Also:

`cos`, `arctan`, `arcsin`

Notes

arccos is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cos(z)* = *x*. The convention is to return the angle *z* whose real part lies in *[0, pi]*.

For real-valued input data types, *arccos* 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, *arccos* 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 *cos* is also known as *acos* or cos^-1.

References

Wikipedia, "Inverse trigonometric function", http://en.wikipedia.org/wiki/Arccos

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```

arctan (*x, [out]*)

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 `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctan* is a complex analytical function that has branch cuts *[1j, infj]* and *[-1j, -infj]* and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as *atan* or `tan^-1`.

References

Wikipedia, "Inverse trigonometric function", http://en.wikipedia.org/wiki/Arctan

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:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```

hypot (*x1, x2, [out]*)

Given two sides of a right triangle, return its hypotenuse.

Parameters

x : array_like

Base of the triangle.

y : array_like

Height of the triangle.

Returns

z : ndarray

Hypotenuse of the triangle: sqrt(x**2 + y**2)

Examples

```
>>> np.hypot(3,4)
5.0
```

arctan2 (*x1, x2, [out]*)

Elementwise arc tangent of `x1/x2` choosing the quadrant correctly.

The quadrant (ie. branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the line segments `(0,0) - (1,0)` and `(0,0) - (x2,x1)`. This function is defined also for *x2* = 0.

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

arctan2 is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard [2]:

x1	*x2*	*arctan2(x1,x2)*
+/- 0	+0	+/- 0
+/- 0	-0	+/- pi
> 0	+/-inf	+0 / +pi
< 0	+/-inf	-0 / -pi
+/-inf	+inf	+/- (pi/4)
+/-inf	-inf	+/- (3*pi/4)

Note that +0 and -0 are distinct floating point numbers.

References

ISO/IEC standard 9899:1999, “Programming language C”, 1999.

Examples

Consider four points in different quadrants:

```
>>> 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 *x2* = 0 and at several other special points, obtaining values in the range `[-pi, pi]`:

```
>>> 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 (*x, [out]*)

Convert angles from radians to degrees.

See Also:

rad2deg
equivalent function; see for documentation.

radians (*x, [out]*)

Convert angles from degrees to radians.

See Also:

deg2rad
equivalent function; see for documentation.

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
Maximum discontinuity between values.

axis : integer
Axis along which unwrap will operate.

Returns

out : ndarray
Output array

3.13.2 Hyperbolic functions

sinh(x[,out])	Hyperbolic sine, element-wise.
cosh(x[,out])	Hyperbolic cosine, element-wise.
tanh(x[,out])	Hyperbolic tangent element-wise.
arcsinh(x[,out])	Inverse hyperbolic sine elementwise.
arccosh(x[,out])	Inverse hyperbolic cosine, elementwise.
arctanh(x[,out])	Inverse hyperbolic tangent elementwise.

sinh (*x, [out]*)

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.

Returns
- **out** : ndarray
  - Output array of same shape as *x*.

cosh (*x, [out]*)

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 (*x, [out]*)

Hyperbolic tangent element-wise.

Parameters
- **x** : array_like
  - Input array.

Returns

y : ndarray

The corresponding hyperbolic tangent values.

arcsinh (*x, [out]*)

Inverse hyperbolic sine elementwise.

Parameters

x : array_like

Input array.

Returns

out : ndarray

Array of 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

Wikipedia, "Inverse hyperbolic function", http://en.wikipedia.org/wiki/Arcsinh

Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

arccosh (*x, [out]*)

Inverse hyperbolic cosine, elementwise.

Parameters

x : array_like

Input array.

Returns

out : ndarray

Array of the same shape and dtype as *x*.

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, inf]`.

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 *[-inf, 1]* and is continuous from above on it.

References

Wikipedia, "Inverse hyperbolic function", http://en.wikipedia.org/wiki/Arccosh

Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
```

arctanh (*x, [out]*)

Inverse hyperbolic tangent elementwise.

Parameters

x : array_like

Input array.

Returns

out : ndarray

Array of the same shape as *x*.

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, -inf]* and *[1, inf]* 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

Wikipedia, "Inverse hyperbolic function", http://en.wikipedia.org/wiki/Arctanh

Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

3.13.3 Rounding

`around`(a[,decimals,out])	Evenly round to the given number of decimals.
`round_`(a[,decimals,out])	Round an array to the given number of decimals.
`rint`(x[,out])	Round elements of the array to the nearest integer.
`fix`(x[,y])	Round to nearest integer towards zero.
`floor`(x[,out])	Return the floor of the input, element-wise.
`ceil`(x[,out])	Return the ceiling of the input, element-wise.

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.

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 [5] and errors introduced when scaling by powers of ten.

References

"How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?", William Kahan, http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

Examples

```
>>> np.around([.5, 1.5, 2.5, 3.5, 4.5])
array([ 0.,  2.,  2.,  4.,  4.])
>>> np.around([1,2,3,11], decimals=1)
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`(*x, [out]*)

Round elements of the array to the nearest integer.

Parameters

x : array_like

Input array.

[5] "Lecture Notes on the Status of IEEE 754", William Kahan, http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

Returns

out : ndarray

Output array is same shape and type as *x*.

Examples

```
>>> a = [-4.1, -3.6, -2.5, 0.1, 2.5, 3.1, 3.9]
>>> np.rint(a)
array([-4., -4., -2.,  0.,  2.,  3.,  4.])
```

fix(*x, y=None*)

Round to nearest integer towards zero.

Round an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

Parameters

x : array_like

An array of floats to be rounded

y : ndarray, optional

Output array

Returns

out : ndarray of floats

The array of rounded numbers

See Also:

floor
Round downwards

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(*x, [out]*)

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*.

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(*x, [out]*)

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

y : {ndarray, scalar}

The ceiling of each element in *x*.

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.])
```

3.13.4 Sums, products, differences

`prod`(a[,axis,dtype,out])	Return the product of array elements over a given axis.
`sum`(a[,axis,dtype,out])	Return the sum of array elements over a given axis.
`nansum`(a[,axis])	Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
`cumprod`(a[,axis,dtype,out])	Return the cumulative product of elements along a given axis.
`cumsum`(a[,axis,dtype,out])	Return the cumulative sum of the elements along a given axis.
`diff`(a[,n,axis])	Calculate the nth order discrete difference along given axis.
`ediff1d`(ary[,to_end,to_begin])	The differences between consecutive elements of an array.
`gradient`(f,*varargs)	Return the gradient of an N-dimensional array.
`cross`(a,b[,axisa,axisb,axisc,...])	Return the cross product of two (arrays of) vectors.
`trapz`(y[,x,dx,axis])	Integrate along the given axis using the composite trapezoidal rule.

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:

`ndarray.prod`
equivalent method

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x) #random
16
```

Examples

By default, calculate the product of all elements:

```
>>> np.prod([1.,2.])
2.0
```

Even when the input array is two-dimensional:

```
>>> np.prod([[1.,2.],[3.,4.]])
24.0
```

But we can also specify the axis over which to multiply:

```
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

If the type of *x* is unsigned, then the output type is the unsigned platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
```

If *x* is of a signed integer type, then the output type is the default platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == np.int
True
```

sum(*a, axis=None, dtype=None, out=None*)

Return the 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.

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

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```

If the accumulator is too small, overflow occurs:

```
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

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 infinity. 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

```
>>> 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

```
>>> 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.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> a = np.array([1,2,3])
>>> np.cumprod(a) # intermediate results 1, 1*2
...               # total product 1*2*3 = 6
array([1, 2, 6])
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.cumprod(a, dtype=float) # specify type of output
array([   1.,    2.,    6.,   24.,  120.,  720.])
```

The cumulative product for each column (i.e., over the rows of) *a*:

```
>>> np.cumprod(a, axis=0)
array([[ 1,  2,  3],
       [ 4, 10, 18]])
```

The cumulative product for each row (i.e. over the columns of) *a*:

```
>>> np.cumprod(a,axis=1)
array([[  1,   2,   6],
       [  4,  20, 120]])
```

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 or object that can be converted to an array.

axis : int, optional

Axis along which the cumulative sum is computed. The default (*axis* = *None*) is to compute the cumsum over the flattened array. *axis* may be negative, in which case it counts from the last to the first axis.

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.

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

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.cumsum(a)
array([ 1,  3,  6, 10, 15, 21])
>>> np.cumsum(a,dtype=float)     # specifies type of output value(s)
array([  1.,   3.,   6.,  10.,  15.,  21.])
>>> np.cumsum(a,axis=0)      # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a,axis=1)      # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

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

Calculate the nth order discrete difference along given axis.

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.

Returns

out : ndarray

The *n* order differences. The shape of the output is the same as *a* except along *axis* where the dimension is *n* less.

Examples

```
>>> x = np.array([0,1,3,9,5,10])
>>> np.diff(x)
array([ 1,  2,  6, -4,  5])
>>> np.diff(x,n=2)
array([  1,   4, -10,   9])
>>> 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

This array will be flattened before the difference is taken.

to_end : number, optional

If provided, this number will be tacked onto the end of the returned differences.

to_begin : number, optional

If provided, this number will be taked onto the beginning of the returned differences.

Returns

ed : array

The differences. Loosely, this will be (ary[1:] - ary[:-1]).

Notes

When applied to masked arrays, this function drops the mask information if the *to_begin* and/or *to_end* parameters are used

gradient (*f, *varargs*)

Return the gradient of an N-dimensional array.

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

```
>>> np.gradient(np.array([[1,1],[3,4]]))
[array([[ 2.,  3.],
       [ 2.,  3.]]),
 array([[ 0.,  0.],
       [ 1.,  1.]])]
```

cross (*a, b, axisa=-1, axisb=-1, axisc=-1, axis=None*)

Return the cross product of two (arrays of) vectors.

The cross product is performed over the last axis of a and b by default, and can handle axes with dimensions 2 and 3. For a dimension of 2, the z-component of the equivalent three-dimensional cross product is returned.

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.

Examples

```
>>> np.trapz([1,2,3])
>>> 4.0
>>> np.trapz([1,2,3], [4,6,8])
>>> 8.0
```

3.13.5 Exponents and logarithms

`exp`(x[,out])	Calculate the exponential of the elements in the input array.
`expm1`(x[,out])	Return the exponential of the elements in the array minus one.
`log`(x[,out])	Natural logarithm, element-wise.
`log10`(x[,out])	Compute the logarithm in base 10 element-wise.
`log2`(x[,y])	Return the base 2 logarithm.
`log1p`(x[,out])	*log(1 + x)* in base *e*, elementwise.

exp (*x, [out]*)

Calculate the exponential of the elements in the input array.

Parameters

x : array_like

Input values.

Returns

out : ndarray

Element-wise exponential of *x*.

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

M. Abramovitz and I. A. Stegun, "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables," Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm

Examples

Plot the magnitude and phase of `exp(x)` in the complex plane:

```
>>> 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 (*x, [out]*)

Return the exponential of the elements in the array minus one.

Parameters

x : array_like

Input values.

Returns

out : ndarray

Element-wise exponential minus one: `out=exp(x)-1`.

See Also:

log1p

`log(1+x)`, the inverse of expm1.

Notes

This function provides greater precision than using `exp(x)-1` for small values of *x*.

Examples

Since the series expansion of `e**x = 1 + x + x**2/2! + x**3/3! + ...`, for very small *x* we expect that `e**x -1 ~ x + x**2/2`:

```
>>> np.expm1(1e-10)
1.00000000005e-10
>>> np.exp(1e-10) - 1
1.000000082740371e-10
```

log (*x, [out]*)

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*.

Parameters

x : array_like

Input value.

Returns

y : ndarray

The natural logarithm of *x*, element-wise.

See Also:

`log10`, `log2`, `log1p`

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

Wikipedia, "Logarithm". http://en.wikipedia.org/wiki/Logarithm

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([  0.,    1.,    2., -Inf])
```

log10 (*x, [out]*)

Compute the logarithm in base 10 element-wise.

Parameters

x : array_like

Input values.

Returns

y : ndarray

Base-10 logarithm of *x*.

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

Wikipedia, "Logarithm". http://en.wikipedia.org/wiki/Logarithm

Examples

```
>>> np.log10([1.e-15,-3.])
array([-15.,  NaN])
```

log2 (*x, y=None*)

Return the base 2 logarithm.

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* elementwise. NaNs are returned where *x* is negative.

See Also:

`log`, `log1p`, `log10`

Examples

```
>>> np.log2([-1,2,4])
array([ NaN,   1.,   2.])
```

log1p (*x, [out]*)

log(1 + x) in base *e*, elementwise.

Parameters

x : array_like

Input values.

Returns

y : ndarray

Natural logarithm of *1 + x*, elementwise.

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, *log1p* 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, *log1p* is a complex analytical function that has a branch cut *[-inf, -1]* and is continuous from above on it. *log1p* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

Examples

```
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

3.13.6 Other special functions

`i0`(x)	Modified Bessel function of the first kind, order 0.
`sinc`(x)	Return the sinc function.

i0 (*x*)

Modified Bessel function of the first kind, order 0.

Usually denoted I_0.

Parameters
x : array_like, dtype float or complex
Argument of the Bessel function.

Returns
out : ndarray, shape z.shape, dtype z.dtype
The modified Bessel function evaluated at the elements of *x*.

See Also:

`scipy.special.iv`, `scipy.special.ive`

References

Wikipedia, "Bessel function", http://en.wikipedia.org/wiki/Bessel_function

Examples

```
>>> 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 $\sin(\pi x)/(\pi x)$.

Parameters
x : ndarray
Array (possibly multi-dimensional) of values for which to 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

Wikipedia, "Sinc function", http://en.wikipedia.org/wiki/Sinc_function

Examples

```
>>> x = np.arange(-20., 21.)/5.
>>> np.sinc(x)
array([ -3.89804309e-17,  -4.92362781e-02,  -8.40918587e-02,
        -8.90384387e-02,  -5.84680802e-02,   3.89804309e-17,
         6.68206631e-02,   1.16434881e-01,   1.26137788e-01,
         8.50444803e-02,  -3.89804309e-17,  -1.03943254e-01,
        -1.89206682e-01,  -2.16236208e-01,  -1.55914881e-01,
         3.89804309e-17,   2.33872321e-01,   5.04551152e-01,
         7.56826729e-01,   9.35489284e-01,   1.00000000e+00,
         9.35489284e-01,   7.56826729e-01,   5.04551152e-01,
         2.33872321e-01,   3.89804309e-17,  -1.55914881e-01,
        -2.16236208e-01,  -1.89206682e-01,  -1.03943254e-01,
        -3.89804309e-17,   8.50444803e-02,   1.26137788e-01,
         1.16434881e-01,   6.68206631e-02,   3.89804309e-17,
        -5.84680802e-02,  -8.90384387e-02,  -8.40918587e-02,
        -4.92362781e-02,  -3.89804309e-17])
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, 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(sinc(xx))
```

3.13.7 Floating point routines

signbit(x[,out])	Returns element-wise True where signbit is set (less than zero).
frexp(x[,out1,out2])	Split the number, x, into a normalized fraction (y1) and exponent (y2)
ldexp(x1,x2[,out])	Compute y = x1 * 2**x2.

signbit (*x, [out]*)
Returns element-wise True where signbit is set (less than zero).

Parameters
x: array_like :
The input value(s).

Returns
out : array_like, bool
Output.

Examples

```
>>> np.signbit(-1.2)
True
>>> np.signbit(np.array([1, -2.3, 2.1]))
array([False,  True, False], dtype=bool)
```

frexp (*x, [out1, out2]*)
Split the number, x, into a normalized fraction (y1) and exponent (y2)

ldexp (*x1, x2, [out]*)
Compute y = x1 * 2**x2.

3.13.8 Arithmetic operations

add(x1,x2[,out])	Add arguments element-wise.
reciprocal(x[,out])	Return element-wise reciprocal.
negative(x[,out])	Returns an array with the negative of each element of the original array.
multiply(x1,x2[,out])	Multiply arguments elementwise.
divide(x1,x2[,out])	Divide arguments element-wise.
power(x1,x2[,out])	Returns element-wise base array raised to power from second array.
subtract(x1,x2[,out])	Subtract arguments element-wise.
true_divide(x1,x2[,out])	Returns an element-wise, true division of the inputs.
floor_divide(x1,x2[,out])	Return the largest integer smaller or equal to the division of the inputs.
fmod(x1,x2[,out])	Return the remainder of division.
mod(x1,x2[,out])	Returns element-wise remainder of division.
modf(x[,out1,out2])	Return the fractional and integral part of a number.
remainder(x1,x2[,out])	Returns element-wise remainder of division.

add (*x1, x2, [out]*)

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

```
>>> 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 (*x, [out]*)

Return element-wise reciprocal.

Parameters
: **x** : array_like
: Input value.

Returns
: **y** : ndarray
: Return value.

Examples

```
>>> reciprocal(2.)
0.5
>>> reciprocal([1, 2., 3.33])
array([ 1.       ,  0.5      ,  0.3003003])
```

negative (*x, [out]*)
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

```
>>> np.negative([1.,-1.])
array([-1.,  1.])
```

multiply (*x1, x2, [out]*)
Multiply arguments elementwise.

Parameters
: **x1, x2** : array_like
: The arrays to be multiplied.

Returns
: **y** : ndarray
: The product of *x1* and *x2*, elementwise. Returns a scalar if both *x1* and *x2* are scalars.

Notes

Equivalent to *x1* * *x2* in terms of array-broadcasting.

Examples

```
>>> 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.],
       [  0.,   4.,  10.],
       [  0.,   7.,  16.]])
```

divide (*x1, x2, [out]*)
Divide arguments element-wise.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

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

```
>>> 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:

```
>>> 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:

```
>>> 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 *seterr*:

```
>>> 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
```

power (*x1, x2, [out]*)

Returns element-wise base array raised to power from second array.

Raise each base in *x1* to the power of the exponents in *x2*. This requires that *x1* and *x2* must be broadcastable to the same shape.

Parameters

x1 : array_like

The bases.

x2 : array_like

The exponents.

Returns

y : ndarray

The bases in *x1* raised to the exponents in *x2*.

Examples

Cube each element in a list.

```
>>> 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.

```
>>> 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.

```
>>> x2 = np.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> np.power(x1, x2)
array([[ 0,   1,   8, 27, 16,   5],
       [ 0,   1,   8, 27, 16,   5]])
```

subtract (*x1, x2, [out]*)

Subtract arguments element-wise.

Parameters

x1, x2 : array_like

The arrays to be subtracted from each other. If type is 'array_like' the *x1* and *x2* shapes must be identical.

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(*x1, x2, [out]*)

Returns an element-wise, true division of the inputs.

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

x2 : array_like

Divisor

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, '//' will be the floor division operator and '/' will be the true division operator. The 'true_divide(*x1*, *x2*)' function is equivalent to true division in Python.

floor_divide(*x1, x2, [out]*)

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(*x1, x2, [out]*)

Return the remainder of division.

This is the NumPy implementation of the C 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:

`mod`

Modulo operation where the quotient is *floor(x1,x2)*.

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, and the sign of the divisor has no influence on the results.

Examples

```
>>> np.fmod([-3, -2, -1, 1, 2, 3], 2)
array([-1,  0, -1,  1,  0,  1])
```

```
>>> np.mod([-3, -2, -1, 1, 2, 3], 2)
array([1, 0, 1, 1, 0, 1])
```

mod(*x1, x2, [out]*)

Returns element-wise remainder of division.

Computes *x1 - floor(x1/x2)*x2*.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

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 *x2* is 0.

Examples

```
>>> np.remainder([4,7],[2,3])
array([0, 1])
```

modf (*x, [out1, out2]*)

Return the fractional and integral part of a number.

The fractional and integral parts are negative if the given number is negative.

Parameters

x : array_like

Input number.

Returns

y1 : ndarray

Fractional part of *x*.

y2 : ndarray

Integral part of *x*.

Examples

```
>>> np.modf(2.5)
(0.5, 2.0)
>>> np.modf(-.4)
(-0.40000000000000002, -0.0)
```

remainder (*x1, x2, [out]*)

Returns element-wise remainder of division.

Computes *x1 - floor(x1/x2)*x2*.

Parameters

x1 : array_like

Dividend array.

x2 : array_like

Divisor array.

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 *x2* is 0.

Examples

```
>>> np.remainder([4,7],[2,3])
array([0, 1])
```

3.13.9 Handling complex numbers

angle(z[,deg])	Return the angle of the complex argument.
real(val)	Return the real part of the elements of the array.
imag(val)	Return the imaginary part of array.
conj(x[,out])	Return the complex conjugate, element-wise.

angle(*z, deg=0*)

Return the angle of the complex argument.

Parameters

z : array_like

A complex number or sequence of complex numbers.

deg : bool, optional

Return angle in degrees if True, radians if False (default).

Returns

angle : {ndarray, scalar}

The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

See Also:

arctan2

Examples

```
>>> np.angle([1.0, 1.0j, 1+1j])               # in radians
array([ 0.        ,  1.57079633,  0.78539816])
>>> np.angle(1+1j, deg=True)                  # in degrees
45.0
```

real(*val*)

Return the real part of the elements of the array.

Parameters

val : array_like

Input array.

Returns

out : ndarray

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 array.

Parameters

val : array_like

Input array.

Returns

out : ndarray, real or int

Real part of each element, same shape as *val*.

conj (*x, [out]*)

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

```
>>> np.conjugate(1+2j)
(1-2j)
```

3.13.10 Miscellaneous

`convolve`(a,v[,mode])	Returns the discrete, linear convolution of two one-dimensional sequences.
`clip`(a,a_min,a_max[,out])	Clip (limit) the values in an array.
`sqrt`(x[,out])	Return the positive square-root of an array, element-wise.
`square`(x[,out])	Return the element-wise square of the input.
`absolute`(x[,out])	Calculate the absolute value element-wise.
`fabs`(x[,out])	Compute the absolute values elementwise.
`sign`(x[,out])	Returns an element-wise indication of the sign of a number.
`maximum`(x1,x2[,out])	Element-wise maximum of array elements.
`minimum`(x1,x2[,out])	Element-wise minimum of array elements.
`nan_to_num`(x)	Replace nan with zero and inf with large numbers.
`real_if_close`(a[,tol])	If complex input returns a real array if complex parts are close to zero.
`interp`(x,xp,fp[,left,right])	One-dimensional linear interpolation.

convolve(*a, v, mode='full'*)

Returns the discrete, linear convolution of two one-dimensional sequences.

The convolution operator is often seen in signal processing, where it models the effect of a linear time-invariant system on a signal [6]. 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.

'valid':

Mode *valid* returns output of length `max(M, N) - min(M, N) + 1`. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

Returns

out : ndarray

Discrete, linear convolution of *a* and *v*.

[6] Wikipedia, "Convolution", http://en.wikipedia.org/wiki/Convolution.

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

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] f[n - m]$$

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

Examples

Note how the convolution operator flips the second array before "sliding" the two across one another:

```
>>> np.convolve([1, 2, 3], [0, 1, 0.5])
array([ 0. ,  1. ,  2.5,  4. ,  1.5])
```

Only return the middle values of the convolution. Contains boundary effects, where zeros are taken into account:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'same')
array([ 1. ,  2.5,  4. ])
```

The two arrays are of the same length, so there is only one position where they completely overlap:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'valid')
array([ 2.5])
```

`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*.

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

sqrt (*x, [out]*)

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 `[-inf, 0)` and is continuous from above on it.

Examples

```
>>> np.sqrt([1,4,9])
array([ 1.,  2.,  3.])
```

```
>>> np.sqrt([4, -1, -3+4J])
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> np.sqrt([4, -1, numpy.inf])
array([  2.,  NaN,  Inf])
```

square (*x, [out]*)

Return the element-wise square of the input.

Parameters

x : array_like

Input data.

Returns

out : ndarray

Element-wise *x*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

```
>>> np.square([-1j, 1])
array([-1.-0.j,  1.+0.j])
```

`absolute` (*x, [out]*)

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

```
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2,  1.2])
>>> np.absolute(1.2 + 1j)
1.5620499351813308
```

Plot the function over `[-10, 10]`:

```
>>> 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:

```
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10])
>>> plt.show()
```

`fabs` (*x, [out]*)

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.

Returns

y : {ndarray, scalar}

The absolute values of *x*, the returned values are always floats.

See Also:

absolute
Absolute values including *complex* types.

Examples

```
>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([ 1.2,  1.2])
```

sign (*x, [out]*)

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

```
>>> np.sign([-5., 4.5])
array([-1.,  1.])
>>> np.sign(0)
0
```

maximum (*x1, x2, [out]*)

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima.

Parameters

x1, x2 : array_like

The arrays holding the elements to be compared.

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

Notes

Equivalent to `np.where(x1 > x2, x1, x2)` but faster and does proper broadcasting.

Examples

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

minimum(*x1, x2, [out]*)

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima.

Parameters

x1, x2 : array_like

The arrays holding the elements to be compared.

Returns

y : {ndarray, scalar}

The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

See Also:

maximum

element-wise maximum

Notes

Equivalent to `np.where(x1 < x2, x1, x2)` but faster and does proper broadcasting.

Examples

```
>>> np.minimum([2, 3, 4], [1, 5, 2])
array([1, 3, 2])
```

```
>>> np.minimum(np.eye(2), [0.5, 2])
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

nan_to_num(*x*)

Replace nan with zero and inf with large numbers.

Parameters

x : array_like

Input data.

Returns

out : ndarray

Array with the same shape and dtype as *x*. Nan is replaced by zero, and inf (-inf) is replaced by the largest (smallest) floating point value that fits in the output dtype.

Examples

```
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
>>> np.nan_to_num(x)
array([  1.79769313e+308,  -1.79769313e+308,   0.00000000e+000,
        -1.28000000e+002,   1.28000000e+002])
```

real_if_close(*a, tol=100*)

If complex input returns a real array if complex parts are close to zero.

"Close to zero" is defined as *tol* * (machine epsilon of the type for *a*).

Parameters

a : array_like

Input array.

tol : scalar

Tolerance for the complex part of the elements in the array.

Returns

out : ndarray

If *a* is real, the type of *a* is used for the output. If *a* has complex elements, the returned type is float.

See Also:

real, imag, angle

Notes

Machine epsilon varies from machine to machine and between data types but Python floats on most platforms have a machine epsilon equal to 2.2204460492503131e-16. You can use 'np.finfo(np.float).eps' to print out the machine epsilon for floats.

Examples

```
>>> np.finfo(np.float).eps      # DOCTEST +skip
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(*x, xp, fp, left=None, right=None*)

One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function with given values at discrete data-points.

Parameters

x : array_like

The x-coordinates of the interpolated values.

xp : 1-D sequence of floats

The x-coordinates of the data points, must be increasing.

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 : {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:

```
np.all(np.diff(xp) > 0)
```

Examples

```
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([ 3. ,  3. ,  2.5,  0.56,  0. ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0
```

Plot an interpolant to the sine function:

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
>>> plt.plot(xvals, yinterp, '-x')
>>> plt.show()
```

3.14 Functional programming

apply_along_axis(func1d,axis,arr,...)	Apply function to 1-D slices along the given axis.
apply_over_axes(func,a,axes)	Apply a function repeatedly over multiple axes.
vectorize(somefunction[,otypes,doc])	Generalized function class.
frompyfunc()	frompyfunc(func, nin, nout) take an arbitrary python function that takes nin objects as input and returns nout objects and return a universal function (ufunc). This ufunc always returns PyObject arrays
piecewise(x,condlist,funclist,...)	Evaluate a piecewise-defined function.

apply_along_axis (*func1d, axis, arr, *args*)

Apply function to 1-D slices along the given axis.

Execute *func1d(a[i],*args)* where *func1d* takes 1-D arrays, *a* is the input array, and *i* is an integer that varies in order to apply the function along the given axis for each 1-D subarray in *a*.

Parameters

func1d : function

This function should be able to take 1-D arrays. It is applied to 1-D slices of *a* along the specified axis.

axis : integer

Axis along which *func1d* is applied.

a : ndarray

Input array.

args : any

Additional arguments to *func1d*.

Returns

out : ndarray

The output array. The shape of *out* is identical to the shape of *a*, except along the *axis* dimension, whose length is equal to the size of the return value of *func1d*.

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.])
```

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

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
<BLANKLINE>
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Sum over axes 0 and 2. The result has same number of dimensions as the original array:

```
>>> np.apply_over_axes(np.sum, a, [0,2])
array([[[ 60],
        [ 92],
        [124]]])
```

class `vectorize`(*pyfunc, otypes=", doc=None*)

Generalized function class.

Define a vectorized function which takes nested sequence of objects or numpy arrays as inputs and returns a numpy array as output, evaluating the function over successive tuples of the input arrays like the python map function except it uses the broadcasting rules of numpy.

Data-type of output of vectorized is determined by calling the function with the first element of the input. This can be avoided by specifying the otypes argument as either a string of typecode characters or a list of data-types specifiers. There should be one data-type specifier for each output.

Parameters

f : callable

A Python function or method.

Examples

```
>>> def myfunc(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])
```

`frompyfunc`()

frompyfunc(func, nin, nout) take an arbitrary python function that takes nin objects as input and returns nout objects and return a universal function (ufunc). This ufunc always returns PyObject arrays

`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 : (N,) ndarray

The input domain.

condlist : list of M (N,)-shaped boolean 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 the length of *funclist* is M+1, then that extra function is the default value, used wherever all conditions are false.

funclist : list of M or M+1 callables, f(x,*args,**kw), or values

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 value is provided then a constant function (`lambda x:  value`) 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 : dictionary, 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.

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 # x runs from -2.5 to 2.5 in steps of 1
>>> np.piecewise(x, [x < 0, x >= 0.5], [-1,1])
array([-1., -1., -1.,  1.,  1.,  1.])
```

Define the absolute value, which is `-x` for `x <0` and `x` for `x >= 0`.

```
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([ 2.5,  1.5,  0.5,  0.5,  1.5,  2.5])
```

3.15 Polynomials

3.15.1 Basics

poly1d	A one-dimensional polynomial class.
polyval(p,x)	Evaluate a polynomial at specific values.
poly(seq_of_zeros)	Return polynomial coefficients given a sequence of roots.
roots(p)	Return the roots of a polynomial with coefficients given in p.

class poly1d(*c_or_r, r=0, variable=None*)

A one-dimensional polynomial class.

Parameters

c_or_r : array_like

Polynomial coefficients, in decreasing powers. For example, `(1, 2, 3)` implies $x^2 + 2x + 3$. If *r* is set to True, these coefficients specify the polynomial roots (values where the polynomial evaluate to 0) instead.

r : bool, optional

If True, *c_or_r* gives the polynomial roots. Default is False.

Examples

Construct the polynomial $x^2 + 2x + 3$:

```
>>> p = np.poly1d([1, 2, 3])
>>> print np.poly1d(p)
   2
1 x + 2 x + 3
```

Evaluate the polynomial:

```
>>> p(0.5)
4.25
```

Find the roots:

```
>>> p.r
array([-1.+1.41421356j, -1.-1.41421356j])
```

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, substracted, multplied 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])
```

polyval (*p*, *x*)

Evaluate a polynomial at specific values.

If p is of length N, this function returns the value:

p[0]*(x**N-1) + p[1]*(x**N-2) + ... + p[N-2]*x + p[N-1]

If x is a sequence then p(x) will be returned for all elements of x. If x is another polynomial then the composite polynomial p(x) will be returned.

Parameters

p : {array_like, poly1d}

1D array of polynomial coefficients from highest degree to zero or an instance of poly1d.

x : {array_like, poly1d}

A number, a 1D array of numbers, or an instance of poly1d.

Returns

values : {ndarray, poly1d}

If either p or x is an instance of poly1d, then an instance of poly1d is returned, otherwise a 1D array is returned. In the case where x is a poly1d, the result is the composition of the two polynomials, i.e., substitution is used.

See Also:

poly1d
A polynomial class.

Notes

Horner's method is used to evaluate the polynomial. Even so, for polynomials of high degree the values may be inaccurate due to rounding errors. Use carefully.

Examples

```
>>> np.polyval([3,0,1], 5)  # 3 * 5**2 + 0 * 5**1 + 1
76
```

poly(*seq_of_zeros*)
Return polynomial coefficients given a sequence of roots.

Calculate the coefficients of a polynomial given the zeros of the polynomial.

If a square matrix is given, then the coefficients for characteristic equation of the matrix, defined by $\det(\mathbf{A} - \lambda\mathbf{I})$, are returned.

Parameters
seq_of_zeros : ndarray
A sequence of polynomial roots or a square matrix.

Returns
coefs : ndarray
A sequence of polynomial coefficients representing the polynomial
:math:'mathrm{coefs}[0] x^{n-1} + mathrm{coefs}[1] x^{n-2} +
... + mathrm{coefs}[2] x + mathrm{coefs}[n]'

See Also:

numpy.poly1d
A one-dimensional polynomial class.

numpy.roots
Return the roots of the polynomial coefficients in p

numpy.polyfit
Least squares polynomial fit

Examples

Given a sequence of polynomial zeros,

```
>>> b = np.roots([1, 3, 1, 5, 6])
>>> np.poly(b)
array([ 1.,   3.,   1.,   5.,   6.])
```

Given a square matrix,

```
>>> P = np.array([[19, 3], [-2, 26]])
>>> np.poly(P)
array([   1.,   -45.,   500.])
```

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.

Examples

```
>>> coeff = [3.2, 2, 1]
>>> print np.roots(coeff)
[-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, rcond=None, full=False*)

Least squares polynomial fit.

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* to points *(x, y)*. Returns a vector of coefficients *p* that minimises the squared error.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int

Degree of the fitting polynomial

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

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]**n * p[n] + ... + x[0] * p[1] + p[0] = y[0]
x[1]**n * p[n] + ... + x[1] * p[1] + p[0] = y[1]
...
x[k]**n * p[n] + ... + x[k] * 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*.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

Wikipedia, “Polynomial interpolation”, http://en.wikipedia.org/wiki/Polynomial_interpolation

Examples

```
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254])
```

It is convenient to use *poly1d* objects for dealing with polynomials:

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
>>> plt.show()
```

3.15.3 Calculus

`polyder`(p[,m])	Return the derivative of the specified order of a polynomial.
`polyint`(p[,m,k])	Return an antiderivative (indefinite integral) of a polynomial.

`polyder` (*p, m=1*)

Return the derivative of the specified order of a polynomial.

Parameters

p : poly1d or sequence

Polynomial to differentiate. A sequence is interpreted as polynomial coefficients, see *poly1d*.

m : int, optional

Order of differentiation (default: 1)

Returns

der : poly1d

A new polynomial representing the derivative.

See Also:

`polyint`
Anti-derivative of a polynomial.

`poly1d`
Class for one-dimensional polynomials.

Examples

The derivative of the polynomial $x^3 + x^2 + x^1 + 1$ is:

```
>>> p = np.poly1d([1,1,1,1])
>>> p2 = np.polyder(p)
>>> p2
poly1d([3, 2, 1])
```

which evaluates to:

```
>>> p2(2.)
17.0
```

We can verify this, approximating the derivative with `(f(x + h) - f(x))/h`:

```
>>> (p(2. + 0.001) - p(2.)) / 0.001
17.007000999997857
```

The fourth-order derivative of a 3rd-order polynomial is zero:

```
>>> np.polyder(p, 2)
poly1d([6, 2])
>>> np.polyder(p, 3)
poly1d([6])
>>> np.polyder(p, 4)
poly1d([ 0.])
```

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:

```
>>> p = 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:

```
>>> 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:

```
>>> np.polyder(P, 2)(0)
6.0
>>> np.polyder(P, 1)(0)
5.0
>>> P(0)
3.0
```

3.15.4 Arithmetic

polyadd(a1,a2)	Returns sum of two polynomials.
polydiv(u,v)	Returns the quotient and remainder of polynomial division.
polymul(a1,a2)	Returns product of two polynomials represented as sequences.
polysub(a1,a2)	Returns difference from subtraction of two polynomials input as sequences.

polyadd(*a1, a2*)

Returns sum of two polynomials.

Returns sum of polynomials; *a1* + *a2*. Input polynomials are represented as an array_like sequence of terms or a poly1d object.

Parameters

a1 : {array_like, poly1d}

Polynomial as sequence of terms.

a2 : {array_like, poly1d}

Polynomial as sequence of terms.

Returns

out : {ndarray, poly1d}

Array representing the polynomial terms.

See Also:

polyval, polydiv, polymul, polyadd

polydiv(*u, v*)

Returns the quotient and remainder of polynomial division.

The input arrays specify the polynomial terms in turn with a length equal to the polynomial degree plus 1.

Parameters

u : {array_like, poly1d}
: Dividend polynomial.

v : {array_like, poly1d}
: Divisor polynomial.

Returns

q : ndarray
: Polynomial terms of quotient.

r : ndarray
: Remainder of polynomial division.

See Also:

poly, polyadd, polyder, polydiv, polyfit, polyint, polymul, polysub, polyval

Examples

$$\frac{3x^2 + 5x + 2}{2x + 1} = 1.5x + 1.75, remainder 0.25$$

```
>>> x = np.array([3.0, 5.0, 2.0])
>>> y = np.array([2.0, 1.0])
>>> np.polydiv(x, y)
>>> (array([ 1.5 ,  1.75]), array([ 0.25]))
```

polymul(*a1, a2*)

Returns product of two polynomials represented as sequences.

The input arrays specify the polynomial terms in turn with a length equal to the polynomial degree plus 1.

Parameters

a1 : {array_like, poly1d}
: First multiplier polynomial.

a2 : {array_like, poly1d}
: Second multiplier polynomial.

Returns

out : {ndarray, poly1d}
: Product of inputs.

See Also:

poly, polyadd, polyder, polydiv, polyfit, polyint, polysub, polyval

polysub(*a1, a2*)

Returns difference from subtraction of two polynomials input as sequences.

Returns difference of polynomials; *a1* - *a2*. Input polynomials are represented as an array_like sequence of terms or a poly1d object.

Parameters

a1 : {array_like, poly1d}
: Minuend polynomial as sequence of terms.

a2 : {array_like, poly1d}
: Subtrahend polynomial as sequence of terms.

Returns
out : {ndarray, poly1d}
Array representing the polynomial terms.

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 Vandermonde matrix is rank deficient.

exception RankWarning
Issued by polyfit when Vandermonde matrix is rank deficient.

3.16 Financial functions

3.16.1 Simple financial functions

fv(rate,nper,pmt,pv[,when])	Compute the future value.
pv(rate,nper,pmt[,fv,when])	Compute the present value.
npv(rate,values)	Returns the NPV (Net Present Value) of a cash flow series.
pmt(rate,nper,pv[,fv,when])	Compute the payment against loan principal plus interest.
ppmt(rate,per,nper,pv[,fv,when])	Not implemented. Compute the payment against loan principal.
ipmt(rate,per,nper,pv[,fv,when])	Not implemented. Compute the payment portion for loan interest.
irr(values)	Return the Internal Rate of Return (IRR).
mirr(values,finance_rate,...)	Modified internal rate of return.
nper(rate,pmt,pv[,fv,when])	Compute the number of periods.
rate(nper,pmt,pv,fv[,when,guess,tol,...])	Compute the rate of interest per period.

fv (*rate, nper, pmt, pv, when='end'*)
Compute the future value.

Parameters

rate : scalar or array_like of shape(M,)
: Rate of interest as decimal (not per cent) per period

nper : scalar or array_like of shape(M,)
: Number of compounding periods

pmt : scalar or array_like of shape(M,)
: Payment

pv : scalar or array_like of shape(M,)
: Present value

when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
: When payments are due ('begin' (1) or 'end' (0)). Defaults to {'end', 0}.

Returns

out : ndarray
: Future values. If all input is scalar, returns a scalar float. If any input is array_like, returns future values for each input element. If multiple inputs are array_like, they all must have the same shape.

Notes

The future value is computed by solving the equation:

```
fv +
pv*(1+rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) == 0
```

or, when `rate == 0`:

```
fv + pv + pmt * nper == 0
```

Examples

What is the future value after 10 years of saving $100 now, with an additional monthly savings of $100. Assume the interest rate is 5% (annually) compounded monthly?

```
>>> np.fv(0.05/12, 10*12, -100, -100)
15692.928894335748
```

By convention, the negative sign represents cash flow out (i.e. money not available today). Thus, saving $100 a month at 5% annual interest leads to $15,692.93 available to spend in 10 years.

If any input is array_like, returns an array of equal shape. Let's compare different interest rates from the example above.

```
>>> a = np.array((0.05, 0.06, 0.07))/12
>>> np.fv(a, 10*12, -100, -100)
array([ 15692.92889434,  16569.87435405,  17509.44688102])
```

pv (*rate, nper, pmt, fv=0.0, when='end'*)
Compute the present value.

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 `pv` is computed by solving the equation:

```
fv +
pv*(1 + rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) = 0
```

or, when `rate = 0`:

```
fv + pv + pmt * nper = 0
```

npv (*rate, values*)

Returns the NPV (Net Present Value) of a cash flow series.

Parameters

rate : scalar

The discount rate.

values : array_like, shape(M,)

The values of the time series of cash flows. Must be the same increment as the *rate*.

Returns

out : float

The NPV of the input cash flow series *values* at the discount *rate*.

Notes

Returns the result of:

$$\sum_{t=1}^{M} \frac{values_t}{(1 + rate)^t}$$

pmt (*rate, nper, pv, fv=0, when='end'*)

Compute the payment against loan principal plus interest.

Parameters

rate : array_like

Rate of interest (per period)

nper : array_like

Number of compounding periods

pv : array_like

Present value

fv : array_like

Future value

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 `pmt` is computed by solving the equation:

```
fv +
pv*(1 + rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) == 0
```

or, when `rate == 0`:

```
fv + pv + pmt * nper == 0
```

Examples

What would the monthly payment need to be to pay off a $200,000 loan in 15 years at an annual interest rate of 7.5%?

```
>>> np.pmt(0.075/12, 12*15, 200000)
-1854.0247200054619
```

In order to pay-off (i.e. have a future-value of 0) the $200,000 obtained today, a monthly payment of $1,854.02 would be required.

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.

nper : array_like

Number of compounding periods

pv : array_like

Present value

fv : array_like, optional

Future value

when : {{'begin', 1}, {'end', 0}}, {string, int}

When payments are due ('begin' (1) or 'end' (0))

See Also:

`pmt`, `pv`, `ipmt`

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.

`pmt = ppmt + ipmt`

`irr` (*values*)

Return the Internal Rate of Return (IRR).

This is the rate of return that gives a net present value of 0.0.

Parameters

values : array_like, shape(N,)

Input cash flows per time period. At least the first value would be negative to represent the investment in the project.

Returns

out : float

Internal Rate of Return for periodic input values.

Examples

```
>>> np.irr([-100, 39, 59, 55, 20])
0.2809484211599611
```

`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.

finance_rate : scalar

Interest rate paid on the cash flows

reinvest_rate : scalar

Interest rate received on the cash flows upon reinvestment

Returns

out : float

Modified internal rate of return

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
fv : array_like, optional
Future value
when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
When payments are due ('begin' (1) or 'end' (0))

Notes

The number of periods `nper` is computed by solving the equation:

```
fv + pv*(1+rate)**nper + pmt*(1+rate*when)/rate * ((1+rate)**nper - 1) == 0
```

or, when `rate == 0`:

```
fv + pv + pmt * 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?

```
>>> 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.

```
>>> 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]],
<BLANKLINE>
       [[ 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 `rate` is computed by solving the equation:

```
fv + pv*(1+rate)**nper + pmt*(1+rate*when)/rate * ((1+rate)**nper - 1) = 0
```

or, if `rate = 0`:

```
fv + pv + pmt * nper = 0
```

3.17 Set routines

3.17.1 Making proper sets

`unique1d`(ar1[,return_index,return_inverse])	Find the unique elements of an array.

unique1d(*ar1, return_index=False, return_inverse=False*)
    Find the unique elements of an array.

Parameters

ar1 : array_like
    This array will be flattened if it is not already 1-D.

return_index : bool, optional
    If True, also return the indices against *ar1* that result in the unique array.

return_inverse : bool, optional
    If True, also return the indices against the unique array that result in *ar1*.

Returns

unique : ndarray
    The unique values.

unique_indices : ndarray, optional
    The indices of the unique values. Only provided if *return_index* is True.

unique_inverse : ndarray, optional
    The indices to reconstruct the original 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

```
>>> np.unique1d([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique1d(a)
array([1, 2, 3])
```

Reconstruct the input from unique values:

```
>>> np.unique1d([1,2,6,4,2,3,2], return_index=True)
>>> x = [1,2,6,4,2,3,2]
>>> u, i = np.unique1d(x, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> i
array([0, 1, 4, 3, 1, 2, 1])
>>> [u[p] for p in i]
[1, 2, 6, 4, 2, 3, 2]
```

3.17.2 Boolean operations

`intersect1d`(ar1,ar2)	Intersection returning repeated or unique elements common to both arrays.
`intersect1d_nu`(ar1,ar2)	Intersection returning unique elements common to both arrays.
`setdiff1d`(ar1,ar2)	Set difference of 1D arrays with unique elements.
`setmember1d`(ar1,ar2)	Return a boolean array set True where first element is in second array.
`setxor1d`(ar1,ar2)	Set exclusive-or of 1D arrays with unique elements.
`union1d`(ar1,ar2)	Union of 1D arrays with unique elements.

`intersect1d`(*ar1, ar2*)

Intersection returning repeated or unique elements common to both arrays.

Parameters

ar1,ar2 : array_like

Input arrays.

Returns

out : ndarray, shape(N,)

Sorted 1D array of common elements with repeating elements.

See Also:

`intersect1d_nu`
Returns only unique common elements.

`numpy.lib.arraysetops`
Module with a number of other functions for performing set operations on arrays.

Examples

```
>>> np.intersect1d([1,3,3],[3,1,1])
array([1, 1, 3, 3])
```

intersect1d_nu(*ar1, ar2*)

Intersection returning unique elements common to both arrays.

Parameters

ar1,ar2 : array_like

Input arrays.

Returns

out : ndarray, shape(N,)

Sorted 1D array of common and unique elements.

See Also:

intersect1d
Returns repeated or unique common elements.

numpy.lib.arraysetops
Module with a number of other functions for performing set operations on arrays.

Examples

```
>>> np.intersect1d_nu([1,3,3],[3,1,1])
array([1, 3])
```

setdiff1d(*ar1, ar2*)

Set difference of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array_like

Input array.

ar2 : array_like

Input comparison array.

Returns

difference : ndarray

The 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.

setmember1d(*ar1, ar2*)

Return a boolean array set True where first element is in second array.

Boolean array is the shape of *ar1* containing True where the elements of *ar1* are in *ar2* and False otherwise.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array_like

Input array.

ar2 : array_like
: Input array.

Returns

mask : ndarray, bool
: 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.

`setxor1d`(*ar1, ar2*)

Set exclusive-or of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array_like
: Input array.

ar2 : array_like
: Input array.

Returns

xor : ndarray
: The values that are only in one, but not both, of the input arrays.

See Also:

`numpy.lib.arraysetops`
: Module with a number of other functions for performing set operations on arrays.

`union1d`(*ar1, ar2*)

Union of 1D arrays with unique elements.

Use unique1d() to generate arrays with only unique elements to use as inputs to this function.

Parameters

ar1 : array_like, shape(M,)
: Input array.

ar2 : array_like, shape(N,)
: Input array.

Returns

union : ndarray
: Unique union of input arrays.

See Also:

`numpy.lib.arraysetops`
: Module with a number of other functions for performing set operations on arrays.

3.18 Window functions

3.18.1 Various windows

`bartlett`(M)	Return the Bartlett window.
`blackman`(M)	Return the Blackman window.
`hamming`(M)	Return the Hamming window.
`hanning`(M)	Return the Hanning window.
`kaiser`(M,beta)	Return the Kaiser window.

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

E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.

Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

Examples

```
>>> np.bartlett(12)
array([ 0.        ,  0.18181818,  0.36363636,  0.54545455,  0.72727273,
        0.90909091,  0.90909091,  0.72727273,  0.54545455,  0.36363636,
        0.18181818,  0.        ])
```

Plot the window and its frequency response (requires SciPy and matplotlib):

```
>>> from numpy import clip, log10, array, bartlett
>>> from numpy.fft import fft
>>> import matplotlib.pyplot as plt
```

```
>>> window = bartlett(51)
>>> plt.plot(window)
>>> plt.title("Bartlett 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 Bartlett window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight'); plt.show()
```

blackman(*M*)

Return the Blackman window.

The Blackman window is a taper formed by using the 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

Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

Examples

```
>>> 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:

```
>>> from numpy import clip, log10, array, bartlett
>>> 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()
```

```
>>> 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 Bartlett window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight'); plt.show()
```

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

$$w(n) = 0.54 + 0.46cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

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

E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.

W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 425.

Examples

```
>>> from numpy import hamming
>>> hamming(12)
array([ 0.08      ,  0.15302337,  0.34890909,  0.60546483,  0.84123594,
        0.98136677,  0.98136677,  0.84123594,  0.60546483,  0.34890909,
        0.15302337,  0.08      ])
```

Plot the window and the frequency response:

```
>>> from numpy import clip, log10, array, hamming
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = hamming(51)
>>> plt.plot(window)
>>> plt.title("Hamming 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 Hamming window")
>>> plt.ylabel("Magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.axis('tight'); plt.show()
```

hanning(*M*)

Return the Hanning window.

The Hanning window is a taper formed by using a weighted cosine.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

Returns

out : ndarray, shape(M,)

The window, 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

$$w(n) = 0.5 - 0.5cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

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

E.R. Kanasewich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 106-108.

W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 425.

Examples

```
>>> 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:

```
>>> from numpy.fft import fft, fftshift
>>> import matplotlib.pyplot as plt

>>> window = np.hanning(51)
>>> plt.subplot(121)
>>> plt.plot(window)
>>> plt.title("Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")

>>> 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.subplot(122)
>>> 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()
```

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.

beta	Window shape
0	Rectangular
5	Similar to a Hamming
6	Similar to a Hanning
8.6	Similar to a Blackman

A beta value of 14 is probably a good starting point. Note that as beta gets large, the window narrows, and so the number of samples needs to be large enough to sample the increasingly narrow spike, otherwise nans will get returned.

Most references to the Kaiser window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 177-178.

Examples

```
>>> 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:

```
>>> from numpy import clip, log10, array, kaiser
>>> 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	with errstate(**state): –> operations in following block use given state.

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'}, 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.

The default is not to change the current behavior.

divide : {'ignore', 'warn', 'raise', 'call'}, optional

Treatment for division by zero.

over : {'ignore', 'warn', 'raise', 'call'}, optional

Treatment for floating-point overflow.

under : {'ignore', 'warn', 'raise', 'call'}, optional

Treatment for floating-point underflow.

invalid : {'ignore', 'warn', 'raise', 'call'}, 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

Set mode:

```
>>> seterr(over='raise') # doctest: +SKIP
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore',
 'under': 'ignore'}
```

```
>>> old_settings = seterr(all='warn', over='raise') # doctest: +SKIP
```

```
>>> int16(32000) * int16(3) # doctest: +SKIP
Traceback (most recent call last):
      File "<stdin>", line 1, in ?
FloatingPointError: overflow encountered in short_scalars
>>> seterr(all='ignore') # doctest: +SKIP
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore',
 'under': 'ignore'}
```

geterr()

Get the current way of handling floating-point errors.

Returns a dictionary with entries "divide", "over", "under", and "invalid", whose values are from the strings "ignore", "print", "log", "warn", "raise", and "call".

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

log_func_or_obj : 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 invalid over under invalid]
```

In other words, `flags = divide + 2*over + 4*under + 8*invalid`.

If an object is provided, it's write method should take one argument, a string.

Returns

h : callable or log instance

The old error handler.

Examples

Callback upon error:

```
>>> def err_handler(type, flag):
    print "Floating point error (%s), with flag %s" % (type, flag)
...

>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')

>>> np.array([1,2,3])/0.0
Floating point error (divide by zero), with flag 1
array([ Inf,  Inf,  Inf])

>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)
```

Log error message:

```
>>> class Log(object):
        def write(self, msg):
            print "LOG: %s" % msg
...

>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')
>>> np.array([1,2,3])/0.0
LOG: Warning: divide by zero encountered in divide
```

```
>>> np.seterrcall(saved_handler)
>>> np.seterr(**save_err)
```

geterrcall()

Return the current callback function used on floating-point errors.

class **errstate**(***kwargs*)

with errstate(**state): –> operations in following block use given state.

# Set error handling to known state. >>> _ = np.seterr(invalid='raise', divide='raise', over='raise', ... under='ignore')

```
>>> a = -np.arange(3)
>>> with np.errstate(invalid='ignore'): # doctest: +SKIP
...     print np.sqrt(a)                # with statement requires Python 2.5
[ 0.     -1.#IND -1.#IND]
>>> print np.sqrt(a.astype(complex))
[ 0.+0.j          0.+1.j          0.+1.41421356j]
>>> print np.sqrt(a)
Traceback (most recent call last):
 ...
FloatingPointError: invalid value encountered in sqrt
>>> with np.errstate(divide='ignore'):  # doctest: +SKIP
...     print a/0
[0 0 0]
>>> print a/0
Traceback (most recent call last):
    ...
FloatingPointError: divide by zero encountered in divide
```

3.19.2 Internal functions

seterrobj(errobj)	Used internally by *seterr*.
geterrobj()	Used internally by *geterr*.

seterrobj(*errobj*)

Used internally by *seterr*.

Parameters

errobj : list

[buffer_size, error_mask, callback_func]

See Also:

seterrcall

geterrobj()

Used internally by *geterr*.

Returns

errobj : list

Internal numpy buffer size, error mask, error callback function.

3.20 Masked array operations

3.20.1 Constants

`ma.MaskType`	

class **MaskType** ()

3.20.2 Creation

From existing data

`ma.masked_array`	Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.
`ma.array`(data[,dtype,copy,order,...])	Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.
`ma.copy`()	copy a.copy(order='C')
`ma.frombuffer`(buffer[,dtype,count,offset])	Interpret a buffer as a 1-dimensional array.
`ma.fromfunction`(function,shape,**kwargs)	Construct an array by executing a function over each coordinate.
`ma.MaskedArray.copy`([order])	Return a copy of the array.

class **masked_array** ()

Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction:

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

Parameters

data : {var}

Input data.

mask : {nomask, sequence}, optional

Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., 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 : {False, True}, optional

Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, False}, optional

Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}, optional

Minimum number of dimensions

fill_value : {var}, optional

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

keep_mask : {True, boolean}, optional

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

hard_mask : {False, boolean}, optional

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

shrink : {True, boolean}, optional

Whether to force compression of an empty mask.

`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*)

Arrays with possibly masked values. Masked values of True exclude the corresponding element from any computation.

Construction:

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

Parameters

data : {var}

Input data.

mask : {nomask, sequence}, optional

Mask. Must be convertible to an array of booleans with the same shape as data: True indicates a masked (eg., 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 : {False, True}, optional

Whether to copy the input data (True), or to use a reference instead. Note: data are NOT copied by default.

subok : {True, False}, optional

Whether to return a subclass of MaskedArray (if possible) or a plain MaskedArray.

ndmin : {0, int}, optional

Minimum number of dimensions

fill_value : {var}, optional

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

keep_mask : {True, boolean}, optional

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

hard_mask : {False, boolean}, optional

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

shrink : {True, boolean}, optional

Whether to force compression of an empty mask.

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(*buffer, dtype=float, count=-1, offset=0*)
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.:

```
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

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

fn : 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]]`. *fn* 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 *fn*.

dtype : data-type, optional

Data-type of the coordinate arrays passed to *fn*. By default, *dtype* is float.

See Also:

`indices`, `meshgrid`

Notes

Keywords other than *shape* and *dtype* are passed to the 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]])
```

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
```

Ones and zeros

`ma.empty`(shape[,dtype,order])	Return a new array of given shape and type, without initialising entries.
`ma.empty_like`(a)	Create a new array with the same shape and type as another.
`ma.masked_all`(shape[,dtype])	Return an empty masked array of the given shape and dtype, where all the data are masked.
`ma.masked_all_like`(arr)	Return an empty masked array of the same shape and dtype as the array *a*, where all the data are masked.
`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 (*shape, dtype=float, order='C'*)

Return a new array of given shape and type, without initialising entries.

Parameters

shape : {tuple of int, 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`

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],  #random data
       [  2.13182611e-314,   3.06959433e-309]])
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],  #random data
       [  496041986,    19249760]])
```

empty_like(*a*)
Create a new array with the same shape and type as another.

Parameters
a : ndarray
Returned array will have same shape and type as *a*.

See Also:

`zeros_like`, `ones_like`, `zeros`, `ones`, `empty`

Notes

This function does *not* initialize the returned array; to do that use *zeros_like* or *ones_like* instead.

Examples

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.empty_like(a)
>>> np.empty_like(a)
array([[-1073741821, -1067702173,       65538],    #random data
       [      25670,    23454291,       71800]])
```

masked_all(*shape, dtype=<type 'float'>*)
Return an empty masked array of the given shape and dtype, where all the data are masked.

Parameters
dtype : dtype, optional
Data type of the output.

masked_all_like(*arr*)
Return an empty masked array of the same shape and dtype as the array *a*, where all the data are masked.

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.]])
```

zeros(*shape, dtype=float, order='C'*)

Return a new array of given shape and type, filled with zeros.

Parameters

shape : {tuple of ints, int}

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:

`numpy.zeros_like`
Return an array of zeros with shape and type of input.

`numpy.ones_like`
Return an array of ones with shape and type of input.

`numpy.empty_like`
Return an empty array with shape and type of input.

`numpy.ones`
Return a new array setting values to one.

`numpy.empty`
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')])
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

3.20.3 Inspecting the array

`ma.all`(self[,axis,out])	Check if all of the elements of *a* are true.
`ma.any`(self[,axis,out])	Check if any of the elements of *a* are true.
`ma.count`(a[,axis])	Count the non-masked elements of the array along the given axis.
`ma.count_masked`(arr[,axis])	Count the number of masked elements along the given axis.
`ma.getmask`(a)	Return the mask of a, if any, or nomask.
`ma.getmaskarray`(arr)	Return the mask of arr, if any, or a boolean array of the shape of a, full of False.
`ma.getdata`(a[,subok])	Return the *_data* part of *a* if *a* is a MaskedArray, or *a* itself.
`ma.nonzero`(self)	Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.
`ma.shape`(obj)	Return the shape of an array.
`ma.size`(obj[,axis])	Return the number of elements along a given axis.
`ma.MaskedArray.data`	
`ma.MaskedArray.mask`	Mask
`ma.MaskedArray.recordmask`	
`ma.MaskedArray.all`(self[,axis,out])	Check if all of the elements of *a* are true.
`ma.MaskedArray.any`(self[,axis,out])	Check if any of the elements of *a* are true.
`ma.MaskedArray.count`(self[,axis])	Count the non-masked elements of the array along the given axis.
`ma.MaskedArray.nonzero`(self)	Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.
`ma.shape`(obj)	Return the shape of an array.
`ma.size`(obj[,axis])	Return the number of elements along a given axis.

all(*self, 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

```
>>> 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 (*self, 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 (*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 the non masked elements are counted.

Returns

result : MaskedArray

A masked array where the mask is True where all data are masked. If axis is None, returns either a scalar ot the masked singleton if all values are masked.

count_masked (*arr, axis=None*)

Count the number of masked elements along the given axis.

Parameters

axis : int, optional

Axis along which to count. If None (default), a flattened version of the array is used.

getmask (*a*)

Return the mask of a, if any, or nomask.

To get a full array of booleans of the same shape as a, use getmaskarray.

getmaskarray (*arr*)
Return the mask of arr, if any, or a boolean array of the shape of a, full of False.

getdata (*a, subok=True*)
Return the *_data* part of *a* if *a* is a MaskedArray, or *a* itself.

Parameters
a : array_like
A ndarray or a subclass of.
subok : {True, False}, optional
Whether to force the output to a 'pure' ndarray (False) or to return a subclass of ndarray if approriate (True).

nonzero (*self*)
Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.

There are as many tuples as dimensions of a, each tuple contains 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: `transpose(a.nonzero())`.

The result of this is always a 2d array, with a row for each non-zero element.

shape (*obj*)
Return the shape of an array.

Parameters
a : array_like
Input array.
Returns
shape : tuple
The elements of the tuple give the lengths of the corresponding array dimensions.

See Also:

`alen`

ndarray.shape
array method

Examples

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1,2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1,2),(3,4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

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
>>> np.size(a,0)
2
```

data

mask
Mask

recordmask

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

```
>>> 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 the non masked elements are counted.
Returns
result : MaskedArray
A masked array where the mask is True where all data are masked. If axis is None, returns either a scalar ot the masked singleton if all values are masked.

nonzero ()
Return the indices of the elements of a that are not zero nor masked, as a tuple of arrays.

There are as many tuples as dimensions of a, each tuple contains 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: `transpose(a.nonzero())`.

The result of this is always a 2d array, with a row for each non-zero element.

shape (*obj*)
Return the shape of an array.

Parameters
a : array_like
Input array.
Returns
shape : tuple
The elements of the tuple give the lengths of the corresponding array dimensions.

See Also:

`alen`

`ndarray.shape`
array method

Examples

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1,2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1,2),(3,4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

`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
>>> np.size(a,0)
2
```

3.20.4 Manipulating a MaskedArray

Changing the shape

`ma.ravel`(self)	Returns a 1D version of self, as a view.
`ma.reshape`(a,new_shape[,order])	Change the shape of the array a to new_shape.
`ma.resize`(x,new_shape)	Return a new array with the specified shape.
`ma.MaskedArray.flatten`([order])	Collapse an array into one dimension.
`ma.MaskedArray.ravel`(self)	Returns a 1D version of self, as a view.
`ma.MaskedArray.reshape`(self,*s,**kwargs)	Returns a masked array containing the data of a, but with a new shape. The result is a view to the original array; if this is not possible, a ValueError is raised.
`ma.MaskedArray.resize`(self,newshape[,refcheck,order])	Change shape and size of array in-place.

ravel (*self*)

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

```
>>> 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'*)

Change the shape of the array a to new_shape.

resize (*x, new_shape*)

Return a new array with the specified shape.

The total size of the original array can be any size. The new array is filled with repeated copies of a. If a was masked, the new array will be masked, and the new mask will be a repetition of the old one.

flatten (*order='C'*)

Collapse an array 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.

Examples

```
>>> 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

```
>>> 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*)

Returns a masked array containing the data of a, but with a new shape. The result is a view to the original array; if this is not possible, a ValueError is raised.

Parameters

shape : shape tuple or int

The new shape should be compatible with the original shape. If an integer, then the result will be a 1D array of that length.

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 : array

A new view to the array.

Notes

If you want to modify the shape in place, please use `a.shape = s`

Examples

```
>>> 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*)
Change shape and size of array in-place.

Modifying axes

`ma.swapaxes`()	swapaxes a.swapaxes(axis1, axis2)
`ma.transpose`(a[,axes])	Return a view of the array with dimensions permuted according to axes, as a masked 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 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

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*)
Return a view of the array with dimensions permuted according to axes, as a masked array.

If `axes` is None (default), the output view has reversed dimensions compared to the original.

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 (**axes*)
Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

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

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

Changing the number of dimensions

`ma.atleast_1d`(*arys)	Convert inputs to arrays with at least one dimension.
`ma.atleast_2d`(*arys)	View inputs as arrays with at least two dimensions.
`ma.atleast_3d`(*arys)	View inputs as arrays with at least three dimensions.
`ma.expand_dims`(x,axis)	Expand the shape of the array by including a new axis before the given one.
`ma.squeeze`(a)	Remove single-dimensional entries from the shape of an array.
`ma.MaskedArray.squeeze`()	Remove single-dimensional entries from the shape of *a*.
`ma.column_stack`(tup)	Stack 1-D arrays as columns into a 2-D array
`ma.concatenate`(arrays[,axis])	Concatenate the arrays along the given axis.
`ma.dstack`(tup)	Stack arrays in sequence depth wise (along third axis)
`ma.hstack`(tup)	Stack arrays in sequence horizontally (column wise)
`ma.hsplit`(ary,indices_or_sections)	Split array into multiple sub-arrays horizontally.
`ma.mr_`	Translate slice objects to concatenation along the first axis.
`ma.row_stack`(tup)	Stack arrays vertically.
`ma.vstack`(tup)	Stack arrays vertically.

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

```
>>> 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

```
>>> numpy.atleast_2d(3.0)
array([[ 3.]])
```

```
>>> x = numpy.arange(3.0)
>>> numpy.atleast_2d(x)
array([[ 0.,  1.,  2.]])
>>> numpy.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 one-dimensional array of shape `N` becomes a view of shape `(1, N, 1)`. An `(M, N)` array becomes a view of shape `(N, M, 1)`.

See Also:

`numpy.atleast_1d`, `numpy.atleast_2d`

Examples

```
>>> numpy.atleast_3d(3.0)
array([[[ 3.]]])
```

```
>>> x = numpy.arange(3.0)
>>> numpy.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = numpy.arange(12.0).reshape(4,3)
>>> numpy.atleast_3d(x).shape
(4, 3, 1)
>>> numpy.atleast_3d(x).base is x
True
```

```
>>> for arr in np.atleast_3d(1, [1, 2], [[1, 2]]): print arr, "\n"
...
[[[1]]]
```

[[[1]

[2]]]

[[[1]

[2]]]

`expand_dims` (*x, axis*)

Expand the shape of the array by including a new axis before the given one.

`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

```
>>> 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(*tup*)

Stack 1-D arrays as columns into a 2-D array

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters
tup : sequence of 1-D or 2-D arrays.
Arrays to stack. All of them must have the same first dimension.

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 the arrays along the given axis.

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

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.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
<BLANKLINE>
       [[2, 3]],
<BLANKLINE>
       [[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 along first axis.

dstack
Stack along third axis.

concatenate
: Join arrays.

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(*ary, indices_or_sections*)

Split array into multiple sub-arrays horizontally.

Please refer to the `numpy.split` documentation. *hsplit* is equivalent to `numpy.split` with `axis = 1`.

See Also:

split
: Split array into multiple sub-arrays.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> np.hsplit(x, 2)
<BLANKLINE>
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [ 12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [ 10.,  11.],
       [ 14.,  15.]])]
```

```
>>> np.hsplit(x, array([3, 6]))
<BLANKLINE>
[array([[  0.,   1.,   2.],
       [  4.,   5.,   6.],
       [  8.,   9.,  10.],
       [ 12.,  13.,  14.]]),
 array([[  3.],
        [  7.],
        [ 11.],
        [ 15.]]),
 array([], dtype=float64)]
```

mr_()

Translate slice objects to concatenation along the first axis.

Examples

```
>>> 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(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
Split an array into a list of multiple sub-arrays of near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

vsplit
Split array into a list of multiple sub-arrays vertically.

dsplit
Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

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]])
```

vstack(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
Split an array into a list of multiple sub-arrays of near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

vsplit
Split array into a list of multiple sub-arrays vertically.

dsplit
Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

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]])
```

Joining arrays

`ma.column_stack`(tup)	Stack 1-D arrays as columns into a 2-D array
`ma.concatenate`(arrays[,axis])	Concatenate the arrays along the given axis.
`ma.dstack`(tup)	Stack arrays in sequence depth wise (along third axis)
`ma.hstack`(tup)	Stack arrays in sequence horizontally (column wise)
`ma.vstack`(tup)	Stack arrays vertically.

column_stack(*tup*)

Stack 1-D arrays as columns into a 2-D array

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters

tup : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

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 the arrays along the given axis.

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

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.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
<BLANKLINE>
       [[2, 3]],
<BLANKLINE>
       [[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 along first axis.

dstack
Stack along third axis.

concatenate
Join arrays.

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]])
```

vstack(*tup*)

Stack arrays vertically.

vstack can be used to rebuild arrays divided by *vsplit*.

Parameters

tup : sequence of arrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

See Also:

array_split
Split an array into a list of multiple sub-arrays of near-equal size.

split
Split array into a list of multiple sub-arrays of equal size.

vsplit
Split array into a list of multiple sub-arrays vertically.

dsplit
Split array into a list of multiple sub-arrays along the 3rd axis (depth).

concatenate
Join arrays together.

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

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

`ma.make_mask`(m[,copy,shrink,flag,...])	Return m as a mask, creating a copy if necessary or requested.
`ma.make_mask_none`(newshape[,dtype])	Return a mask of shape s, filled with False.
`ma.mask_or`(m1,m2[,copy,shrink])	Return the combination of two masks m1 and m2.
`ma.make_mask_descr`(ndtype)	Constructs a dtype description list from a given dtype. Each field is set to a bool.

make_mask(*m, copy=False, shrink=True, flag=None, dtype=<type 'numpy.bool_'>*)
Return m as a mask, creating a copy if necessary or requested.

The function can accept any sequence of integers or nomask. Does not check that contents must be 0s and 1s.

Parameters
m : array_like
Potential mask.
copy : bool
Whether to return a copy of m (True) or m itself (False).
shrink : bool

Whether to shrink m to nomask if all its values are False.

dtype : dtype

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.

make_mask_none (*newshape, dtype=None*)

Return a mask of shape s, filled with False.

Parameters

news : tuple

A tuple indicating the shape of the final mask.

dtype: {None, dtype}, optional :

If None, use MaskType. Otherwise, use a new datatype with the same fields as *dtype* with boolean type.

mask_or (*m1, m2, copy=False, shrink=True*)

Return the combination of two masks m1 and m2.

The masks are combined with the *logical_or* operator, treating nomask as False. The result may equal m1 or m2 if the other is nomask.

Parameters

m1 : array_like

First mask.

m2 : array_like

Second mask

copy : {False, True}, optional

Whether to return a copy.

shrink : {True, False}, optional

Whether to shrink m to nomask if all its values are False.

Raises

ValueError :

If m1 and m2 have different flexible dtypes.

make_mask_descr (*ndtype*)

Constructs a dtype description list from a given dtype. Each field is set to a bool.

Accessing a mask

`ma.getmask`(a)	Return the mask of a, if any, or nomask.
`ma.getmaskarray`(arr)	Return the mask of arr, if any, or a boolean array of the shape of a, full of False.
`ma.masked_array.mask`	Mask

getmask (*a*)

Return the mask of a, if any, or nomask.

To get a full array of booleans of the same shape as a, use getmaskarray.

getmaskarray (*arr*)

Return the mask of arr, if any, or a boolean array of the shape of a, full of False.

mask

Mask

Finding masked data

`ma.flatnotmasked_contiguous`(a)	Find contiguous unmasked data in a flattened masked array.
`ma.flatnotmasked_edges`(a)	Find the indices of the first and last not masked values in a 1D masked array. If all values are masked, returns None.
`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 not masked values along the given axis in a masked array.

`flatnotmasked_contiguous`(*a*)

Find contiguous unmasked data in a flattened masked array.

Return a sorted sequence of slices (start index, end index).

`flatnotmasked_edges`(*a*)

Find the indices of the first and last not masked values in a 1D masked array. If all values are masked, returns None.

`notmasked_contiguous`(*a, axis=None*)

Find contiguous unmasked data in a masked array along the given axis.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns

A sorted sequence of slices (start index, end index). :

Notes

Only accepts 2D arrays at most.

`notmasked_edges`(*a, axis=None*)

Find the indices of the first and last not masked values along the given axis in a masked array.

If all values are masked, return None. Otherwise, return a list of 2 tuples, corresponding to the indices of the first and last unmasked values respectively.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

Modifying a mask

`ma.mask_cols`(a[,axis])	Mask whole columns of a 2D array that contain masked values.
`ma.mask_or`(m1,m2[,copy,shrink])	Return the combination of two masks m1 and m2.
`ma.mask_rowcols`(a[,axis])	Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected with the *axis* parameter.
`ma.mask_rows`(a[,axis])	Mask whole rows of a 2D array that contain masked values.
`ma.harden_mask`()	harden_mask(self) Force the mask to hard.
`ma.soften_mask`()	soften_mask(self) Force the mask to soft.
`ma.MaskedArray.harden_mask`(self)	Force the mask to hard.
`ma.MaskedArray.soften_mask`(self)	Force the mask to soft.
`ma.MaskedArray.shrink_mask`(self)	Reduce a mask to nomask when possible.
`ma.MaskedArray.unshare_mask`(self)	Copy the mask and set the sharedmask flag to False.

mask_cols (*a, axis=None*)

Mask whole columns of a 2D array that contain masked values.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

mask_or (*m1, m2, copy=False, shrink=True*)

Return the combination of two masks m1 and m2.

The masks are combined with the *logical_or* operator, treating nomask as False. The result may equal m1 or m2 if the other is nomask.

Parameters

m1 : array_like

First mask.

m2 : array_like

Second mask

copy : {False, True}, optional

Whether to return a copy.

shrink : {True, False}, optional

Whether to shrink m to nomask if all its values are False.

Raises

ValueError :

If m1 and m2 have different flexible dtypes.

mask_rowcols (*a, axis=None*)

Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected with 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
: **axis** : int, optional
: Axis along which to perform the operation. If None, applies to a flattened version of the array.

Returns
: **a *pure* ndarray.** :

`mask_rows`(*a, axis=None*)
Mask whole rows of a 2D array that contain masked values.

Parameters
: **axis** : int, optional
: Axis along which to perform the operation. If None, applies to a flattened version of the array.

`harden_mask`()
harden_mask(self) Force the mask to hard.

`soften_mask`()
soften_mask(self) Force the mask to soft.

`harden_mask`()
Force the mask to hard.

`soften_mask`()
Force the mask to soft.

`shrink_mask`()
Reduce a mask to nomask when possible.

`unshare_mask`()
Copy the mask and set the sharedmask flag to False.

3.20.6 Conversion operations

> to a masked array

`ma.asarray`(a[,dtype,order])	Convert the input *a* to a masked array of the given datatype.
`ma.asanyarray`(a[,dtype])	Convert the input *a* to a masked array of the given datatype. If *a* is a subclass of MaskedArray, its class is conserved.
`ma.fix_invalid`(a[,mask,copy,fill_value])	Return (a copy of) *a* where invalid data (nan/inf) are masked and replaced by *fill_value*.
`ma.masked_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x == value)`.
`ma.masked_greater`(x,value[,copy])	Return the array *x* masked where `(x > value)`. Any value of mask already masked is kept masked.
`ma.masked_greater_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x >= value)`.
`ma.masked_inside`(x,v1,v2[,copy])	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.
`ma.masked_invalid`(a[,copy])	Mask the array for invalid values (NaNs or infs). Any preexisting mask is conserved.
`ma.masked_less`(x,value[,copy])	Shortcut to masked_where, with condition `(x < value)`.
`ma.masked_less_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x <= value)`.
`ma.masked_not_equal`(x,value[,copy])	Shortcut to masked_where, with condition `(x != value)`.
`ma.masked_object`(x,value[,copy,shrink])	Mask the array x where the data are exactly equal to value.
`ma.masked_outside`(x,v1,v2[,copy])	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.
`ma.masked_values`(x,value[,rtol,atol,copy,shrink])	Mask the array x where the data are approximately equal in value, i.e. `(abs(x - value) <= atol+rtol*abs(value))`
`ma.masked_where`(condition,a[,copy])	Return an array masked where `condition` is `True`. Masked values of `a` or `condition` are kept.

asarray(*a, dtype=None, order=None*)
Convert the input *a* to a masked array of the given datatype.

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 ('FORTRAN') memory representation. Defaults to 'C'.

Returns

out : ndarray

MaskedArray interpretation of *a*. No copy is performed if the input is already an ndarray. If *a* is a subclass of MaskedArray, a base class MaskedArray is returned.

asanyarray(*a, dtype=None*)

Convert the input *a* to a masked array of the given datatype. If *a* is a subclass of MaskedArray, its class is conserved.

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 ('FORTRAN') memory representation. Defaults to 'C'.

Returns

out : ndarray

MaskedArray interpretation of *a*. No copy is performed if the input is already an ndarray.

fix_invalid(*a, mask=False, copy=True, fill_value=None*)

Return (a copy of) *a* where invalid data (nan/inf) are masked and replaced by *fill_value*.

Note that a copy is performed by default (just in case...).

Parameters

a : array_like

A (subclass of) ndarray.

copy : bool

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

fill_value : {var}, optional

Value used for fixing invalid data. If not given, the output of get_fill_value(a) is used instead.

Returns

b : MaskedArray

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

Shortcut to masked_where, with condition `(x == value)`.

See Also:

`masked_where`
base function

`masked_values`
equivalent function for floats.

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

Return the array *x* masked where `(x > value)`. Any value of mask already masked is kept masked.

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

Shortcut to masked_where, with condition `(x >= value)`.

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

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.

Notes

The array x is prefilled with its filling value.

`masked_invalid`(*a, copy=True*)
Mask the array for invalid values (NaNs or infs). Any preexisting mask is conserved.

`masked_less`(*x, value, copy=True*)
Shortcut to masked_where, with condition `(x < value)`.

`masked_less_equal`(*x, value, copy=True*)
Shortcut to masked_where, with condition `(x <= value)`.

`masked_not_equal`(*x, value, copy=True*)
Shortcut to masked_where, with condition `(x != value)`.

`masked_object`(*x, value, copy=True, shrink=True*)
Mask the array *x* where the data are exactly equal to value.

This function is suitable only for object arrays: for floating point, please use **'masked_values'_** instead.

Parameters

x : array_like
Array to mask

value : var
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

`masked_outside`(*x, v1, v2, copy=True*)
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.

Notes

The array x is prefilled with its filling value.

`masked_values`(*x, value, rtol=1.0000000000000001e-05, atol=1e-08, copy=True, shrink=True*)
Mask the array x where the data are approximately equal in value, i.e. `(abs(x - value) <= atol+rtol*abs(value))`

Suitable only for floating points. For integers, please use `masked_equal`. The mask is set to `nomask` if posible.

Parameters

x : array_like
Array to fill.

value : float
Masking value.

rtol : {float}, optional
Tolerance parameter.

atol : {float}, optional
Tolerance parameter (1e-8).

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

masked_where (*condition, a, copy=True*)
Return `a` as an array masked where `condition` is `True`. Masked values of `a` or `condition` are kept.

Parameters
condition : array_like
Masking condition.
a : array_like
Array to mask.
copy : bool
Whether to return a copy of `a` (True) or modify `a` in place (False).

> to a ndarray

`ma.compress_cols`(a)	Suppress whole columns of a 2D array that contain masked values.
`ma.compress_rowcols`(x[,axis])	Suppress the rows and/or columns of a 2D array that contain masked values.
`ma.compress_rows`(a)	Suppress whole rows of a 2D array that contain masked values.
`ma.compressed`(x)	Return a 1-D array of all the non-masked data.
`ma.filled`(a[,fill_value])	Return *a* as an array where masked data have been replaced by *value*.
`ma.MaskedArray.compressed`(self)	Return a 1-D array of all the non-masked data.
`ma.MaskedArray.filled`(self[,fill_value])	Return a copy of self, where masked values are filled with *fill_value*.

compress_cols (*a*)
Suppress whole columns of a 2D array that contain masked values.

compress_rowcols (*x, axis=None*)
Suppress the rows and/or columns of a 2D array that contain masked values.

The suppression behavior is selected with the *axis* parameter.

- If axis is None, 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. If None, applies to a flattened version of the array.
Returns
compressed_array : an ndarray.

compress_rows (*a*)
Suppress whole rows of a 2D array that contain masked values.

compressed(*x*)
Return a 1-D array of all the non-masked data.

See Also:

MaskedArray.compressed
equivalent method

filled(*a, fill_value=None*)
Return *a* as an array where masked data have been replaced by *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 : {var}, optional
Filling value. If None, the output of `get_fill_value(a)` is used instead.
Returns
a : array_like

compressed()
Return a 1-D array of all the non-masked data.

Returns
data : ndarray.
A new ndarray holding the non-masked data is returned.

Notes

- The result is NOT a MaskedArray !

Examples

```
>>> x = array(arange(5), mask=[0]+[1]*4)
>>> print x.compressed()
[0]
>>> print type(x.compressed())
<type 'numpy.ndarray'>
```

filled(*fill_value=None*)
Return a copy of self, where masked values are filled with *fill_value*.

If *fill_value* is None, *self.fill_value* is used instead.

Notes

- Subclassing is preserved
- The result is NOT a MaskedArray !

Examples

```
>>> 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'>
```

> to another object

`ma.MaskedArray.tofile`(self,fid[,sep,format])	
`ma.MaskedArray.tolist`(self[,fill_value])	Copy the data portion of the array to a hierarchical python list and returns that list.
`ma.MaskedArray.torecords`(self)	Transforms a MaskedArray into a flexible-type array with two fields:
`ma.MaskedArray.tostring`(self[,fill_value,order])	Return a copy of array data as a Python string containing the raw bytes in the array. The array is filled beforehand.

tofile (*fid, sep='', format='%s'*)

tolist (*fill_value=None*)

Copy the data portion of the array to a hierarchical python list and returns that 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`.

torecords ()

Transforms a MaskedArray into a flexible-type array with two fields:

- the _`data` field stores the _`data` part of the array;
- the _`mask` field stores the _`mask` part of the array;

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

```
>>> 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.torecords()
[[(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 a copy of array data as a Python string containing the raw bytes in the array. The array is filled beforehand.

Parameters

fill_value : {var}, optional

Value used to fill in the masked values. If None, uses self.fill_value instead.

order : {string}

Order of the data item in the copy {'C','F','A'}. 'C' – C order (row major) 'Fortran' – Fortran order (column major) 'Any' – Current order of array. None – Same as "Any"

Notes

As for method:*ndarray.tostring*, information about the shape, dtype..., but also fill_value will be lost.

Pickling and unpickling

`ma.dump`(a,F)	Pickle the MaskedArray *a* to the file *F*. *F* can either be the handle of an exiting file, or a string representing a file name.
`ma.dumps`(a)	Return a string corresponding to the pickling of the MaskedArray.
`ma.load`(F)	Wrapper around `cPickle.load` which accepts either a file-like object or a filename.
`ma.loads`(strg)	Load a pickle from the current string.

`dump`(*a, F*)
Pickle the MaskedArray *a* to the file *F*. *F* can either be the handle of an exiting file, or a string representing a file name.

`dumps`(*a*)
Return a string corresponding to the pickling of the MaskedArray.

`load`(*F*)
Wrapper around `cPickle.load` which accepts either a file-like object or a filename.

`loads`(*strg*)
Load a pickle from the current string.

Filling a masked array

`ma.common_fill_value`(a,b)	Return the common filling value of a and b, if any. If a and b have different filling values, returns None.
`ma.default_fill_value`(obj)	Calculate the default fill value for the argument object.
`ma.maximum_fill_value`(obj)	Calculate the default fill value suitable for taking the maximum of `obj`.
`ma.maximum_fill_value`(obj)	Calculate the default fill value suitable for taking the maximum of `obj`.
`ma.set_fill_value`(a,fill_value)	Set the filling value of a, if a is a masked array. Otherwise, do nothing.
`ma.MaskedArray.get_fill_value`(self)	Return the filling value.
`ma.MaskedArray.set_fill_value`(self[, value])	Set the filling value to value.
`ma.MaskedArray.fill_value`	Filling value.

`common_fill_value`(*a, b*)
Return the common filling value of a and b, if any. If a and b have different filling values, returns None.

`default_fill_value`(*obj*)
Calculate the default fill value for the argument object.

`maximum_fill_value`(*obj*)
Calculate the default fill value suitable for taking the maximum of `obj`.

`maximum_fill_value`(*obj*)
Calculate the default fill value suitable for taking the maximum of `obj`.

`set_fill_value`(*a, fill_value*)
Set the filling value of a, if a is a masked array. Otherwise, do nothing.

Parameters
a : ndarray
Input array
fill_value : var
Filling value. A consistency test is performed to make sure the value is compatible with the dtype of a.

Returns
None :

`get_fill_value`()
Return the filling value.

`set_fill_value`(*value=None*)
Set the filling value to value.

If value is None, use a default based on the data type.

`fill_value`
Filling value.

3.20.7 Masked arrays arithmetics

Arithmetics

ma.anom(self[,axis,dtype])	Return the anomalies (deviations from the average) along the given axis.
ma.anomalies(self[,axis,dtype])	Return the anomalies (deviations from the average) along the given axis.
ma.average(a[,axis,weights,returned])	Average the array over the given axis.
ma.conjugate(x[,out])	Return the complex conjugate, element-wise.
ma.corrcoef(x[,y,rowvar,bias,...])	The correlation coefficients formed from the array x, where the rows are the observations, and the columns are variables.
ma.cov(x[,y,rowvar,bias,...])	Estimates the covariance matrix.
ma.cumsum(self[,axis,dtype,out])	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,dtype,out])	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(self[,axis,dtype,out])	Returns the average of the array elements along given axis. Refer to numpy.mean for full documentation.
ma.median(a[,axis,out,overwrite_input])	Compute the median along the specified axis.
ma.power(a,b[,third])	Computes a**b elementwise.
ma.prod(self[,axis,dtype,out])	Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.
ma.std(self[,axis,dtype,out,...])	Compute the standard deviation along the specified axis.
ma.sum(self[,axis,dtype,out])	Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.
ma.var(self[,axis,dtype,out,...])	Compute the variance along the specified axis.
ma.MaskedArray.anom(self[,axis,dtype])	Return the anomalies (deviations from the average) along the given axis.
ma.MaskedArray.cumprod(self[,axis,dtype,out])	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.cumsum(self[,axis,dtype,out])	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.mean(self[,axis,dtype,out])	Returns the average of the array elements along given axis. Refer to numpy.mean for full documentation.
ma.MaskedArray.prod(self[,axis,dtype,out])	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,dtype,out,...])	Compute the standard deviation along the specified axis.

anom(*self, axis=None, dtype=None*)
Return the anomalies (deviations from the average) along the given axis.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.
dtype : {dtype}, optional
Datatype for the intermediary computation. If not given, the current dtype is used instead.

anomalies(*self, axis=None, dtype=None*)
Return the anomalies (deviations from the average) along the given axis.

Parameters
axis : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.
dtype : {dtype}, optional
Datatype for the intermediary computation. If not given, the current dtype is used instead.

average(*a, axis=None, weights=None, returned=False*)
Average the array over the given axis.

Parameters
axis : {None,int}, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.
weights : {None, sequence}, optional
Sequence of weights. The weights must have the shape of a, or be 1D with length the size of a along the given axis. If no weights are given, weights are assumed to be 1.
returned : {False, True}, optional
Flag indicating whether a tuple (result, sum of weights/counts) should be returned as output (True), or just the result (False).

conjugate(*x, [out]*)
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

```
>>> np.conjugate(1+2j)
(1-2j)
```

corrcoef(*x, y=None, rowvar=True, bias=False, allow_masked=True*)
The correlation coefficients formed from the array x, where the rows are the observations, and the columns are variables.

corrcoef(x,y) where x and y are 1d arrays is the same as corrcoef(transpose([x,y]))

Parameters

x : ndarray

Input data. If x is a 1D array, returns the variance. If x is a 2D array, returns the covariance matrix.

y : {None, ndarray} optional

Optional set of variables.

rowvar : {False, True} optional

If True, then each row is a variable with observations in columns. If False, each column is a variable and the observations are in the rows.

bias : {False, True} optional

Whether to use a biased (True) or unbiased (False) estimate of the covariance. If True, then the normalization is by N, the number of non-missing observations. Otherwise, the normalization is by (N-1).

allow_masked : {True, False} 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:

`cov`

`cov` (*x, y=None, rowvar=True, bias=False, allow_masked=True*)

Estimates the covariance matrix.

Normalization is by (N-1) where N is the number of observations (unbiased estimate). If bias is True then normalization is by N.

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

Input data. If x is a 1D array, returns the variance. If x is a 2D array, returns the covariance matrix.

y : array_like, optional

Optional set of variables.

rowvar : {False, True} optional

If rowvar is true, then each row is a variable with observations in columns. If rowvar is False, each column is a variable and the observations are in the rows.

bias : {False, True} optional

Whether to use a biased (True) or unbiased (False) estimate of the covariance. If bias is True, then the normalization is by N, the number of observations. Otherwise, the normalization is by (N-1).

allow_masked : {True, False} 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.

`cumsum` (*self, 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.

Warning: The mask is lost if out is not a valid `MaskedArray` !

Notes

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]
```

cumprod(*self, 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

Arithmetic is modular when using integer types, and no error is raised on overflow.

mean (*self, axis=None, dtype=None, out=None*)

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

See Also:

`numpy.mean`
equivalent function'

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 (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 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 is returned unless out is specified, in which case a reference to out is returned. Return datatype is float64 for ints and floats smaller than float64, or the input datatype 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.

power (*a, b, third=None*)

Computes a**b elementwise.

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.

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.])
```

`std`(*self, 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 (biased estimate).

Returns

standard_deviation : {ndarray, scalar}; 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:

numpy.var
Variance

numpy.mean
Average

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `var = sqrt(mean(abs(x - x.mean())**2))`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, 0)
array([ 1.,  1.])
>>> np.std(a, 1)
array([ 0.5,  0.5])
```

sum(*self, 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

```
>>> 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 (*self, 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

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of 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.

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])
```

anom(*axis=None, dtype=None*)

Return the anomalies (deviations from the average) along the given axis.

Parameters

axis : int, optional

Axis along which to perform the operation. If None, applies to a flattened version of the array.

dtype : {dtype}, optional

Datatype for the intermediary computation. If not given, the current dtype is used instead.

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

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.

Warning: The mask is lost if out is not a valid `MaskedArray` !

Notes

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 along given axis. Refer to `numpy.mean` for full documentation.

See Also:

`numpy.mean`
equivalent function'

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.])
```

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 (biased estimate).

Returns

standard_deviation : {ndarray, scalar}; 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:

numpy.var
Variance

numpy.mean
Average

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `var = sqrt(mean(abs(x - x.mean())**2))`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, 0)
array([ 1.,  1.])
>>> np.std(a, 1)
array([ 0.5,  0.5])
```

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

```
>>> 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.

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

Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of 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.

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])
```

Minimum/maximum

`ma.argmax`(a[,axis,fill_value])	Function version of the eponymous method.
`ma.argmin`(a[,axis,fill_value])	Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.
`ma.max`(obj[,axis,out,fill_value])	Return the maximum along a given axis.
`ma.min`(obj[,axis,out,fill_value])	Return the minimum along a given axis.
`ma.ptp`(obj[,axis,out,fill_value])	Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).
`ma.MaskedArray.argmax`(self[,axis,fill_value,out])	Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.
`ma.MaskedArray.argmin`(self[,axis,fill_value,out])	Return array of indices to the minimum values along the given axis.
`ma.MaskedArray.max`(self[,axis,out,fill_value])	Return the maximum along a given axis.
`ma.MaskedArray.min`(self[,axis,out,fill_value])	Return the minimum along a given axis.
`ma.MaskedArray.ptp`(self[,axis,out,fill_value])	Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

argmax (*a, axis=None, fill_value=None*)
Function version of the eponymous method.

argmin (*a, axis=None, fill_value=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])
```

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

```
>>> 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]
```

`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. If not given, 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.

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 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:

kind	speed	worst case	work space	stable
‘quicksort’	1	O(n^2)	0	no
‘mergesort’	2	O(n*log(n))	~n/2	yes
‘heapsort’	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```
>>> 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:

```
>>> 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. If not given, 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.

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:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Examples

```
>>> 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:

```
>>> 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')])
```

Algebra

`ma.diag`(v[,k])	Extract a diagonal or construct a diagonal array.
`ma.dot`(a,b[,strict])	Return the dot product of two 2D masked arrays a and b.
`ma.identity`(n[,dtype])	Return the identity array.
`ma.inner`(a,b)	Inner product of two arrays.
`ma.innerproduct`(a,b)	Inner product of two arrays.
`ma.outer`(a,b)	Returns the outer product of two vectors.
`ma.outerproduct`(a,b)	Returns the outer product of two vectors.
`ma.trace`(self[,offset,axis1,axis2,...])	Return the sum along diagonals of the array.
`ma.transpose`(a[,axes])	Return a view of the array with dimensions permuted according to axes, as a masked array.
`ma.MaskedArray.trace`([offset,axis1,axis2,...])	Return the sum along diagonals of the array.
`ma.MaskedArray.transpose`(*axes)	Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

diag(*v, k=0*)

Extract a diagonal or construct a diagonal array.

Parameters

v : array_like

If *v* is a 2-dimensional array, return a copy of its *k*-th diagonal. If *v* is a 1-dimensional array, return a 2-dimensional array with *v* on the *k*-th diagonal.

k : int, optional

Diagonal in question. The defaults is 0.

Examples

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

dot(*a, b, strict=False*)

Return the dot product of two 2D masked arrays a and b.

Like the generic numpy equivalent, the product sum is over the last dimension of a and the second-to-last dimension of b. If strict is True, masked values are propagated: if a masked value appears in a row or column, the whole row or column is considered masked.

Parameters

strict : {boolean}

Whether masked data are propagated (True) or set to 0 for the computation.

Notes

The first argument is not conjugated.

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.]])
```

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:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

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:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

outer (*a, b*)

Returns the outer product of two vectors.

Given two vectors, `[a0, a1, ..., aM]` and `[b0, b1, ..., bN]`, the outer product becomes:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

Parameters

a : array_like, shaped (M,)

First input vector. If either of the input vectors are not 1-dimensional, they are flattened.

b : array_like, shaped (N,)

Second input vector.

Returns

out : ndarray, shaped (M, N)

`out[i, j] = a[i] * b[j]`

Notes

Masked values are replaced by 0.

Examples

```
>>> 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*)

Returns the outer product of two vectors.

Given two vectors, `[a0, a1, ..., aM]` and `[b0, b1, ..., bN]`, the outer product becomes:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

Parameters

a : array_like, shaped (M,)
: First input vector. If either of the input vectors are not 1-dimensional, they are flattened.

b : array_like, shaped (N,)
: Second input vector.

Returns

out : ndarray, shaped (M, N)
: `out[i, j] = a[i] * b[j]`

Notes

Masked values are replaced by 0.

Examples

```
>>> 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(*self, offset=0, axis1=0, axis2=1, dtype=None, out=None) a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

numpy.trace
: equivalent function

transpose(*a, axes=None*)

Return a view of the array with dimensions permuted according to axes, as a masked array.

If `axes` is None (default), the output view has reversed dimensions compared to the original.

trace(*offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

numpy.trace
: equivalent function

transpose(**axes*)

Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

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

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`.

Parameters

x : array_like

Input array.

N : int, optional

Order of (number of columns in) the output.

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.

Examples

```
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

polyfit (*x, y, deg, rcond=None, full=False*)

Least squares polynomial fit.

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* to points *(x, y)*. Returns a vector of coefficients *p* that minimises the squared error.

Parameters

x : array_like, shape (M,)

x-coordinates of the M sample points `(x[i], y[i])`.

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int

Degree of the fitting polynomial

rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

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

Wikipedia, “Polynomial interpolation”, http://en.wikipedia.org/wiki/Polynomial_interpolation

Examples

```
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254])
```

It is convenient to use *poly1d* objects for dealing with polynomials:

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
/... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
>>> plt.show()
```

Clipping and rounding

`ma.around`()	Round an array to the given number of decimals.
`ma.clip`(a,a_min,a_max[,out])	Clip (limit) the values in an array.
`ma.round`(a[,decimals,out])	Return a copy of a, rounded to 'decimals' places.
`ma.MaskedArray.clip`(a_min,a_max[,out])	Return an array whose values are limited to `[a_min, a_max]`.
`ma.MaskedArray.round`([decimals,out])	Return an array rounded a to the given number of decimals.

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*.

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, [3,4,1,1,1,4,4,4,4,4], 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(*a_min, a_max, out=None*)

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,...])	Returns True if two arrays are element-wise equal within a tolerance.
ma.apply_along_axis(func1d,axis,arr,...)	Apply function to 1-D slices along the given axis.
ma.arange([start,]stop[,step,...])	Return evenly spaced values within a given interval.
ma.choose(a,choices[,out,mode])	Use an index array to construct a new array from a set of choices.
ma.ediff1d(arr[,to_end,to_begin])	Computes 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)	Returns a (subclass of) masked array, shaped like condition, where the elements are x when condition is True, and y otherwise. If neither x nor y are given, returns a tuple of indices where condition is True (a la condition.nonzero()).

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.

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.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * *b*) and the absolute difference (*atol*) are added together to compare against the absolute difference between *a* and *b*.

Parameters

a, b : array_like
    Input arrays to compare.

fill_value : boolean, optional
    Whether masked values in a or b are considered equal (True) or not (False).

rtol : Relative tolerance
    The relative difference is equal to *rtol* * *b*.

atol : Absolute tolerance
    The absolute difference is equal to *atol*.

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`, `alltrue`, `sometrue`

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.

`apply_along_axis` (*func1d, axis, arr, *args, **kwargs*)

Apply function to 1-D slices along the given axis.

Execute *func1d(a[i],*args)* where *func1d* takes 1-D arrays, *a* is the input array, and *i* is an integer that varies in order to apply the function along the given axis for each 1-D subarray in *a*.

Parameters

func1d : function

This function should be able to take 1-D arrays. It is applied to 1-D slices of *a* along the specified axis.

axis : integer

Axis along which *func1d* is applied.

a : ndarray

Input array.

args : any

Additional arguments to *func1d*.

Returns

out : ndarray

The output array. The shape of *out* is identical to the shape of *a*, except along the *axis* dimension, whose length is equal to the size of the return value of *func1d*.

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.])
```

`arange` (*[start], stop, [step], dtype=None*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in range 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

```
>>> 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`(*a, 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 : int array

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

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

Computes the differences between consecutive elements of an array.

This function is the equivalent of `numpy.ediff1d` that takes masked values into account.

Returns

output : MaskedArray

See Also:

`numpy.eddif1d`
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 : 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]])
```

where (*condition | x, y*)

Returns a (subclass of) masked array, shaped like condition, where the elements are x when condition is True, and y otherwise. If neither x nor y are given, returns a tuple of indices where condition is True (a la condition.nonzero()).

Parameters

condition : {var}

The condition to meet. Must be convertible to an integer array.

x : {var}, optional

Values of the output when the condition is met

y : {var}, optional

Values of the output when the condition is not met.

3.21 Numpy-specific help functions

3.21.1 Finding help

lookfor(what[,module,import_modules,...])	Do a keyword search on docstrings.

lookfor (*what, module=None, import_modules=True, regenerate=False*)

Do a keyword search on docstrings.

A list of of objects that matched the search is displayed, sorted by relevance.

Parameters

what : str

String containing words to look for.

module : str, module

Module whose docstrings to go through.

import_modules : bool

Whether to import sub-modules in packages. Will import only modules in `__all__`.

regenerate : bool

Whether to re-generate the docstring cache.

Examples

```
>>> np.lookfor('binary representation')
Search results for 'binary representation'
------------------------------------------
numpy.binary_repr
    Return the binary representation of the input number as a string.
```

3.21.2 Reading help

info([object,maxwidth,...])	Get help information for a function, class, or module.
source(object[,output,mode'w'at0x2aaaaaac9198>)	Print or write to a file the source code for a Numpy object.

info(*object=None, maxwidth=76, output=<open file '<stdout>', mode 'w' at 0x2aaaaaac9198>, toplevel='numpy'*)
Get help information for a function, class, or module.

Parameters

object : optional
Input object to get information about.

maxwidth : int, optional
Printing width.

output : file like object open for writing, optional
Write into file like object.

toplevel : string, optional
Start search at this level.

Examples

```
>>> np.info(np.polyval) # doctest: +SKIP
```

polyval(p, x)

Evaluate the polymnomial p at x.
...

source(*object, output=<open file '<stdout>', mode 'w' at 0x2aaaaaac9198>*)
Print or write to a file the source code for a Numpy object.

Parameters

object : numpy object
Input object.

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.

3.22 Miscellaneous routines

3.22.1 Buffer objects

getbuffer(obj[,offset[,size]])	Create a buffer object from the given object referencing a slice of length size starting at offset. Default is the entire buffer. A read-write buffer is attempted followed by a read-only buffer.
newbuffer(size)	Return a new uninitialized buffer object of size bytes

getbuffer(*obj, [offset, [size]]*)
Create a buffer object from the given object referencing a slice of length size starting at offset. Default is the entire buffer. A read-write buffer is attempted followed by a read-only buffer.

newbuffer(*size*)
Return a new uninitialized buffer object of size bytes

3.22.2 Performance tuning

`alterdot`()	Change *dot*, *vdot*, and *innerproduct* to use accelerated BLAS functions.
`restoredot`()	Restore *dot*, *vdot*, and *innerproduct* to the default non-BLAS implementations.
`setbufsize`(size)	Set the size of the buffer used in ufuncs.
`getbufsize`()	Return the size of the buffer used in ufuncs.

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
restoredot 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 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:

```
>>> import math
>>> from numpy.lib import scimath
>>> scimath.log(-math.exp(1)) == (1+1j*math.pi)
True
```

Similarly, sqrt(), other base logarithms, power() and trig functions are correctly handled. See their respective docstrings for specific examples.

3.24 Matrix library (`numpy.matlib`)

This module contains all functions in the `numpy` namespace, with the following replacement functions that return `matrices` instead of `ndarrays`.

3.25 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.

3.25.1 Linear algebra

`cholesky`(a)	Cholesky decomposition.
`det`(a)	Compute the determinant of an array.
`eig`(a)	Compute eigenvalues and right eigenvectors of an array.
`eigh`(a[,UPLO])	Eigenvalues and eigenvectors of a Hermitian or real symmetric matrix.
`eigvals`(a)	Compute the eigenvalues of a general matrix.
`eigvalsh`(a[,UPLO])	Compute the eigenvalues of a Hermitean or real symmetric matrix.
`inv`(a)	Compute the inverse of a matrix.
`lstsq`(a,b[,rcond])	Return the least-squares solution to an equation.
`norm`(x[,ord])	Matrix or vector norm.
`pinv`(a[,rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
`solve`(a,b)	Solve the equation `a x = b` for `x`.
`svd`(a[,full_matrices,compute_uv])	Singular Value Decomposition.

3.25.2 FFT

`fft`(a[,n,axis])	Compute the one dimensional fft on a given axis.
`fft2`(a[,s,axes,-1))	Compute the 2-D FFT of an array.
`fftn`(a[,s,axes])	Compute the N-dimensional Fast Fourier Transform.
`ifft`(a[,n,axis])	Compute the one-dimensonal inverse fft along an axis.
`ifft2`(a[,s,axes,-1))	Compute the inverse 2d fft of an array.
`ifftn`(a[,s,axes])	Compute the inverse of fftn.

3.25.3 Other

`i0`(x)	Modified Bessel function of the first kind, order 0.

3.26 Numarray compatibility (`numpy.numarray`)

3.27 Old Numeric compatibility (`numpy.oldnumeric`)

3.28 C-Types Foreign Function Interface (`numpy.ctypeslib`)

`as_array`(**args, **kwds*)

`as_ctypes`(**args, **kwds*)

`ctypes_load_library`(**args, **kwds*)

`load_library`(**args, **kwds*)

`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 : string or tuple of strings

Array flags; may be one or more of:

- C_CONTIGUOUS / C / CONTIGUOUS
- F_CONTIGUOUS / F / FORTRAN
- OWNDATA / O
- WRITEABLE / W
- ALIGNED / A
- UPDATEIFCOPY / U

Examples

```
>>> clib.somefunc.argtypes = [np.ctypeslib.ndpointer(dtype=float64,
...                                                  ndim=1,
...                                                  flags='C_CONTIGUOUS')]
>>> clib.somefunc(np.array([1, 2, 3], dtype=np.float64))
```

CHAPTER

FOUR

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

`Configuration`	
`get_numpy_include_dirs`()	
get_numarray_include_dirs	
`dict_append`(d,**kws)	
`appendpath`(prefix,path)	
`allpath`(name)	Convert a /-separated pathname to one using the OS's path separator.
`dot_join`(*args)	
`generate_config_py`(target)	Generate config.py file containing system_info information used during building the package.
`get_cmd`(cmdname[,_cache])	
`terminal_has_colors`()	
`red_text`(s)	
`green_text`(s)	
`yellow_text`(s)	
`blue_text`(s)	
`cyan_text`(s)	
`cyg2win32`(path)	
`all_strings`(lst)	Return True if all items in lst are string objects.
`has_f_sources`(sources)	Return True if sources contains Fortran files
`has_cxx_sources`(sources)	Return True if sources contains C++ files
`filter_sources`(sources)	Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.
`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, caller_level=1, setup_name='setup.py', **attrs*)

get_numpy_include_dirs()

dict_append(*d, **kws*)

appendpath(*prefix, path*)

allpath(*name*)
Convert a /-separated pathname to one using the OS's path separator.

dot_join(**args*)

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))

get_cmd(*cmdname, _cache={}*)

terminal_has_colors()

red_text(*s*)

green_text(*s*)

yellow_text(*s*)

blue_text(*s*)

cyan_text(*s*)

cyg2win32(*path*)

all_strings(*lst*)
Return True if all items in lst are string objects.

has_f_sources(*sources*)
Return True if sources contains Fortran files

has_cxx_sources(*sources*)
Return True if sources contains C++ files

filter_sources(*sources*)
Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.

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. Thus, this method may be used as setup(**config.todict()).

get_distribution()

Return the distutils distribution object for self.

get_subpackage(*subpackage_name, subpackage_path=None*)

Return a Configuration instance for the sub-package given. If subpackage_path is 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.

add_subpackage(*subpackage_name, subpackage_path=None*)

Add a sub-package to the current Configuration instance. This is useful in a setup.py script for adding sub-packages to a package. The sub-package is contained in subpackage_path / subpackage_name and this directory may contain a setup.py script or else a default setup (suitable for Python-code-only subpackages) is assumed. If the subpackage_path is None, then it is assumed to be located in the local path / subpackage_name.

add_data_files(**files*)

Add files to the list of data_files to be included with the package. 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. An example may clarify:

```
self.add_data_files('foo.dat',
('fun', ['gun.dat', 'nun/pun.dat', '/tmp/sun.dat']),
'bar/cat.dat',
'/full/path/to/can.dat')
```

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 \mypackage \mysubpackage').

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.

add_data_dir(*data_path*)

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. For example suppose the source directory contains fun/foo.dat and fun/bar/car.dat:

```
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/
      car.dat
  sun/
    foo.dat
    bar/
      car.dat
  gun/
    foo.dat
    car.dat
```

add_include_dirs(**paths*)

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 the given sequence of files to the beginning of the headers list. By default, headers will be installed under <python- include>/<self.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.

add_extension(*name, sources, **kw*)

Create and add an Extension instance to the ext_modules list. The first argument defines the name of the extension module that will be installed under the self.name package. The second argument is a list of sources. This method also takes the following optional keyword arguments that are passed on to the Extension constructor: include_dirs, define_macros, undef_macros, library_dirs, libraries, runtime_library_dirs, extra_objects, swig_opts, depends, language, f2py_options, module_dirs, and extra_info.

The self.paths(...) method is applied to all lists that may contain paths. The extra_info is a dictionary or a list of dictionaries whose content will be appended to the keyword arguments. 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.

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.

`add_library`(*name, sources, **build_info*)
Add a library to the list of libraries. Allowed keyword arguments are depends, macros, include_dirs, extra_compiler_args, and f2py_options. The name is the name of the library to be built and sources is a list of sources (or source generating functions) to add to the library.

`add_scripts`(**files*)
Add the sequence of files to the beginning of the scripts list. Scripts will be installed under the <prefix>/bin/ directory.

`paths`(**paths*)
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`()
True if a Fortran 77 compiler is available (because a simple Fortran 77 code was able to be compiled successfully).

`have_f90c`()
True if a Fortran 90 compiler is available (because a simple Fortran 90 code was able to be compiled successfully)

`get_version`()
Return a version string of the current package or None if the version information could not be detected. 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`()
Appends a data function to the data_files list that will generate __svn_version__.py file to the current package directory. This file will be removed from the source directory when Python exits (so that it can be re-generated next time the package is built). This is intended for working with source directories that are in an SVN repository.

`make_config_py`()
Generate a package __config__.py file containing system information used during the building of the package. This file is installed to the package installation directory.

`get_info`(**names*)
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`(name[,notfound_action])	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`	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

cpu ()

set_verbosity (*v, force=False*)

4.2 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.

4.2.1 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>

4.2.2 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.

CHAPTER

FIVE

NUMPY C-API

Beware of the man who won't be bothered with details.
— *William Feather, Sr.* The truth is out there.
— *Chris Carter, The X Files*

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 then 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`). Additional 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

`PyArray_Type`
The Python type of the ndarray 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

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

```
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 builtin 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' repesents 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 builtin 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`
Indicates that items of this data-type must be reference counted (using `Py_INCREF` and `Py_DECREF`).

`NPY_ITEM_LISTPICKLE`
Indicates arrays of this data-type must be converted to a list before pickling.

`NPY_ITEM_IS_POINTER`
Indicates the item is a pointer to some other data-type

`NPY_NEEDS_INIT`
Indicates memory for this data-type must be initialized (set to 0) on creation.

`NPY_NEEDS_PYAPI`
Indicates this data-type requires the Python C-API during access (so don't give up the GIL if array access is going to be needed).

`NPY_USE_GETITEM`
On array access use the `f->getitem` function pointer instead of the standard conversion to an array scalar. Must use if you don't define an array scalar to go along with the data-type.

`NPY_USE_SETITEM`
When creating a 0-d array from an array scalar use `f->setitem` instead of the standard copy from an array scalar. Must use if you don't define an array scalar to go along with the data-type.

`NPY_FROM_FIELDS`
The bits that are inherited for the parent data-type if these bits are set in any field of the data-type. Currently (`NPY_NEEDS_INIT` | `NPY_LIST_PICKLE` | `NPY_ITEM_REFCOUNT` | `NPY_NEEDS_PYAPI`).

`NPY_OBJECT_DTYPE_FLAGS`
Bits set for the object data-type: (`NPY_LIST_PICKLE` | `NPY_USE_GETITEM` | `NPY_ITEM_IS_POINTER` | `NPY_REFCOUNT` | `NPY_NEEDS_INIT` | `NPY_NEEDS_PYAPI`).

`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;
    PyArray_CompareFunc *compare;
    PyArray_ArgFunc *argmax;
    PyArray_DotFunc *dotfunc;
    PyArray_ScanFunc *scanfunc;
    PyArray_FromStrFunc *fromstr;
    PyArray_NonzeroFunc *nonzero;
    PyArray_FillFunc *fill;
    PyArray_FillWithScalarFunc *fillwithscalar;
    PyArray_SortFunc *sort[PyArray_NSORTS];
    PyArray_ArgSortFunc *argsort[PyArray_NSORTS];
    PyObject *castdict;
    PyArray_ScalarKindFunc *scalarkind;
    int **cancastscalarkindto;
    int *cancastto;
    int listpickle
} PyArray_ArrFuncs;
```

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 mis-behaved 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. The 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`, :data'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 `PyCObject *` 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 **nin**
The number of input arguments.

int **nout**
The number of output arguments.

int **nargs**
The total number of arguments (*nin* + *nout*). This must be less than NPY_MAXARGS.

int **identity**
Either PyUFunc_One, PyUFunc_Zero, or PyUFunc_None to indicate the identity for this operation. It is only used for a reduce-like call on an empty array.

`void` **`functions`**
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 *dims* [0] times. The first argument, *args*, is an array of *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 *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 *extradata*. The size of this function pointer array is ntypes.

`void` **`data`**
Extra data to be passed to the 1-d vector loops or `NULL` if no extra-data is needed. This C-array must be the same size (*i.e.* ntypes) as the functions array. `NULL` is used if 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.

`int` **`ntypes`**
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.

`int` **`check_return`**
Obsolete and unused. However, it is set by the corresponding entry in the main ufunc creation routine: `PyUFunc_FromFuncAndData` (...).

`char` **`name`**
A string name for the ufunc. This is used dynamically to build the __doc__ attribute of ufuncs.

`char` **`types`**
An array of *nargs* $\times$ *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 *ntypes* functions, the corresponding set of type numbers in this array shows how the *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.

`char` **`doc`**
Documentation for the ufunc. Should not contain the function signature as this is generated dynamically when __doc__ is retrieved.

`void` **`ptr`**
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.

`PyObject` **`obj`**
For ufuncs dynamically created from python functions, this member holds a reference to the underlying Python function.

`PyObject` **`userloops`**
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

`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 tp_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]*d *ims_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.

PyArrayFlags_Type

`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

> **Bool**, **Byte**, **Short**, **Int**, **Long**, **LongLong**, **UByte**, **UShort**, **UInt**, **ULong**, **ULongLong**, **Float**, **Double**, **LongDouble**, **CFloat**, **CDouble**, **CLongDouble**, **String**, **Unicode**, **Void**, and **Object**.

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

`PyArray_Dims`

This structure is very useful when shape and/or strides information is supposed to be interpreted. The structure is:

```
typedef struct {
    npy_intp *ptr;
    int len;
} PyArray_Dims;
```

The members of this structure are

`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

`PyArray_Chunk`

This is equivalent to the buffer object structure in Python up to the ptr member. On 32-bit platforms (*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.

```
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 (*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 PyCObject 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 DECREF'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.

```
typedef struct {
    int two;
    int nd;
    char typekind;
    int itemsize;
    int flags;
    npy_intp *shape;
    npy_intp *strides;
    void *data;
    PyObject *descr;
} PyArrayInterface;
```

int **two**
the integer 2 as a sanity check.

int **nd**
the number of dimensions in the array.

char **typekind**
A character indicating what kind of array is present according to the typestring convention with 't' -> bitfield, 'b' -> Boolean, 'i' -> signed integer, 'u' -> unsigned integer, 'f' -> floating point, 'c' -> complex floating point, 'O' -> object, 'S' -> string, 'U' -> unicode, 'V' -> void.

int **itemsize**
The number of bytes each item in the array requires.

int **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).

npy_intp **shape**
An array containing the size of the array in each dimension.

npy_intp **strides**
An array containing the number of bytes to jump to get to the next element in each dimension.

void **data**
A pointer *to* the first element of the array.

PyObject **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_LONGDOUBLE** 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. One of `NPY_BIG_ENDIAN` `NPY_LITTLE_ENDIAN` or is defined, and `NPY_BYTE_ORDER` is either 4321 or 1234.

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 `NPY_{NAME}` where `{NAME}` can be

BOOL, **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**, **CFLOAT**, **CDOUBLE**, **CLONGDOUBLE**, **OBJECT**, **STRING**, **UNICODE**, **VOID**

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 `NPY_{NAME}LTR` where `{NAME}` can be

BOOL, **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**, **CFLOAT**, **CDOUBLE**, **CLONGDOUBLE**, **OBJECT**, **STRING**, **VOID**

INTP, **UINTP**

GENBOOL, **SIGNED**, **UNSIGNED**, **FLOATING**, **COMPLEX**

The latter group of `{NAME}s` corresponds to letters used in the array interface typestring specification.

5.3.2 Defines

Max and min values for integers

NPY_MAX_INT{bits}

NPY_MAX_UINT{bits}

NPY_MIN_INT{bits}
These are defined for `{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).

NPY_MIN_{type}
This is defined for `{type}` = **BYTE**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **INTP**

NPY_MAX_{type}
This is defined for all defined for `{type}` = **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **INTP**, **UINTP**

Number of bits in data types

All `NPY_SIZEOF_{CTYPE}` constants have corresponding `NPY_BITSOF_{CTYPE}` constants defined. The `NPY_BITSOF_{CTYPE}` constants provide the number of bits in the data type. Specifically, the available `{CTYPE}s` are

BOOL, **CHAR**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**

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

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.

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

npy_(c)float
float

npy_(c)double
double

npy_(c)longdouble
long 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

`npy_int{bits}`, `npy_uint{bits}`, `npy_float{bits}`, and `npy_complex{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.

`NPY_LONGLONG_FMT`, `NPY_ULONGLONG_FMT`, `NPY_INTP_FMT`, `NPY_UINTP_FMT`, `NPY_LONGDOUBLE_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).

`void *` **`PyArray_DATA`**(*PyObject *obj*)

`char *` **`PyArray_BYTES`**(*PyObject *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.

`npy_intp *` **`PyArray_DIMS`**(*PyObject *arr*)

`npy_intp *` **`PyArray_STRIDES`**(*PyObject* arr*)

`npy_intp` **`PyArray_DIM`**(*PyObject* arr, int n*)
: Return the shape in the n th dimension.

`npy_intp` **`PyArray_STRIDE`**(*PyObject* arr, int n*)
: Return the stride in the n 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 writeable. 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 `npy_intp`. You may want to typecast the returned pointer to the data type of the ndarray.

5.4.2 Creating arrays

From scratch

`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 *subtype*, which must be a subtype of `PyArray_Type`. The array has *nd* dimensions, described by *dims*. The data-type descriptor of the new array is *descr*. If *subtype* is not `&PyArray_Type` (*e.g.* a Python subclass of the ndarray), then *obj* is the object to pass to the `__array_finalize__` method of the subclass. If *data* is `NULL`, then new memory will be allocated and *flags* can be non-zero to indicate a Fortran-style contiguous array. If *data* is not `NULL`, then it is assumed to point to the memory to be used for the array and the *flags* argument is used as the new flags for the array (except the state of `NPY_OWNDATA` and `UPDATEIFCOPY` flags of the new array will be reset). In addition, if *data* is non-NULL, then *strides* can also be provided. If *strides* is `NULL`, then the array strides are computed as C-style contiguous (default) or Fortran-style contiguous (*flags* is nonzero for *data* = `NULL` or *flags* & `NPY_F_CONTIGUOUS` is nonzero non-NULL *data*). Any provided *dims* and *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 `PyArray_DescrNew` (...) except you specify the data-type descriptor with *type_num* and *itemsize*, where *type_num* corresponds to a builtin (or user-defined) type. If the type always has the same number of bytes, then itemsize is ignored. Otherwise, itemsize specifies the particular size of this array.

Warning: If data is passed to `PyArray_NewFromDescr` or `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 `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 itemsize, and the data of the array.

`PyObject*` **`PyArray_SimpleNew`** (*int nd, npy_intp* dims, int typenum*)
Create a new unitialized array of type, *typenum*, whose size in each of *nd* dimensions is given by the integer array, *dims*. This function cannot be used to create a flexible-type array (no itemsize given).

`PyObject*` **`PyArray_SimpleNewFromData`** (*int nd, npy_intp* dims, int typenum, void* data*)
Create an array wrapper around *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 *dims* c-array of length *nd*. The data-type of the array is indicated by *typenum*.

`PyObject*` **`PyArray_SimpleNewFromDescr`** (*int nd, npy_intp* dims, PyArray_Descr* descr*)
Create a new array with the provided data-type descriptor, *descr* , of the shape deteremined by *nd* and *dims*.

`PyArray_FILLWBYTE` (*PyObject* obj, int val*)
Fill the array pointed to by *obj* —which must be a (subclass of) bigndarray—with the contents of *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 *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. Fill the memory with zeros (or the 0 object if *dtype* corresponds to `PyArray_OBJECT`).

`PyObject*` **`PyArray_ZEROS`** (*int nd, npy_intp* dims, int type_num, int fortran*)
Macro form of `PyArray_Zeros` which takes a type-number instead of a data-type object.

`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 *type*, 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. 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.

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 reaised.

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 | 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

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.

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.

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.

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.

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.

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:

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.

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..

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 typenumber. 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.

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

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 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_ISEXTENDED(*num*)

PyDataType_ISEXTENDED(*descr*)

PyArray_ISEXTENDED(*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.

`PyArray_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 explict 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* op, int* n*)
Convert a sequence of Python objects contained in *op* 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:

```
mps = PyArray_ConvertToCommonType(obj, &n);
if (mps==NULL) return NULL;
{code}
<before return>
for (i=0; i<n; i++) Py_DECREF(mps[i]);
PyDataMem_FREE(mps);
{return}
```

`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 typecode 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 INCREF 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 loacation *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

Basic Array Flags

An ndarray can have a data segment that is not a simple contiguous chunk of well-behaved memory you can manipulate. 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 Fortan-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).

`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.

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*)
Evalutes 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 itemsizes 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* inplace 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`. This argument is used if the number of dimension is (or is being resized to be) greater than 2. 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 $10 \times 20 \times 30$, and *permute* `.ptr` is (0,2,1) the shape of the result is $10 \times 30 \times 20$. If *permute* is `NULL`, the shape of the result is $30 \times 20 \times 10$.

`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_INTP`. If a tuple is returned (nd $\neq$ 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, cumsum, prod, and cumprod 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_Clip`** (*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 $10^{-\text{decimals}}$ 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.

`PyObject*` **`PyArray_Std`** (*PyArrayObject* self, int axis, int rtype, PyArrayObject* out*)
Equivalent to `ndarray.std` (*self*, *axis*, *rtype*). Return the standard deviation using data along *axis* converted to data type *rtype*.

`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 specifice 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.

`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.

`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.

`PyObject*` **`PyArray_InnerProduct`** (*PyObject* obj1, PyObject* obj2*)
Compute a product-sum over the last dimensions of *obj1* and *obj2*. Neither array is conjugated.

`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.

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*.

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).

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

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*)
Incremement 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-iterators)

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 mult-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 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 neededtype 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.

5.4.10 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.

`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 builtin 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-typed descriptor (and all it's 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.

5.4.11 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 compatibile 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*)
Convert Python strings into the corresponding byte-order character: '>', '<', 's', '=', or '|'.

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()
```

npy_intp **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`.

5.4.12 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.

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 intialization 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:

```
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

On the other hand, coolhelper.c would contain at the top:

```
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#define NO_IMPORT_ARRAY
#include numpy/arrayobject.h
```

`unsigned int` **`PyArray_GetNDArrayCVersion`**(*void*)

This just returns the value `NPY_VERSION`. 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.

Internal Flexibility

`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.

`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`.

`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

`char*` **`PyDataMem_NEW`**(*size_t nbytes*)

`PyDataMem_FREE`(*char* ptr*)

`char*` **`PyDataMem_RENEW`** (*void * ptr, size_t newbytes*)
Macros to allocate, free, and reallocate memory. These macros are used internally to create arrays.

`npy_intp*` **`PyDimMem_NEW`** (*nd*)

`PyDimMem_FREE` (*npy_intp* ptr*)

`npy_intp*` **`PyDimMem_RENEW`** (*npy_intp* ptr, npy_intp newnd*)
Macros to allocate, free, and reallocate dimension and strides memory.

`PyArray_malloc` (*nbytes*)

`PyArray_free` (*ptr*)

`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 excecute 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, `NPY_ALLOW_THREADS` is defined to the python-defined `WITH_THREADS` constant unless the environment variable `NPY_NOSMP` is set in which case `NPY_ALLOW_THREADS` is defined to be 0.

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.

`NPY_BEGIN_ALLOW_THREADS`
Equivalent to `Py_BEGIN_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

`NPY_END_ALLOW_THREADS`
Equivalent to `Py_END_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

`NPY_BEGIN_THREADS_DEF`
Place in the variable declaration area. This macro sets up the variable needed for storing the Python state.

`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.

`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.

`NPY_BEGIN_THREADS_DESCR` (*PyArray_Descr *dtype*)

Useful to release the GIL only if *dtype* does not contain arbitrary Python objects which may need the Python interpreter during execution of the loop. Equivalent to

NPY_END_THREADS_DESCR(*PyArray_Descr *dtype*)
Useful to regain the GIL in situations where it was released using the BEGIN form of this macro.

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.

NPY_ALLOW_C_API_DEF
Place in the variable declaration area to set up the necessary variable.

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).

NPY_DISABLE_C_API
Place after code that needs to call the Python C-API (to re-release the GIL).

Tip: Never use semicolons after the threading support macros.

Priority

NPY_PRIOIRTY
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 **DIVIDEBYZERO**, **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, errormask, and errorobj). 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 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.

Parameters

- *nin* – The number of inputs to this operation.
- *nout* – The number of outputs
- *ntypes* – How many different data-type "signatures" the ufunc has implemented.
- *func* – Must to an array of length *ntypes* containing `PyUFuncGenericFunction` items. These items are pointers to functions that acutally implement the underlying (element-by-element) function N times. T
- *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.
- *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.
- *name* – The name for the ufunc.
- *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.

`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.

`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.

`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.

`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-compatibile hardware is determined a -1 is returned, otherwise a 0 is returned.

`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.

`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.

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 **PyUFunc_dd_d**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

void **PyUFunc_gg_g**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

void **PyUFunc_FF_F_As_DD_D**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

void **PyUFunc_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 **PyUFunc_GG_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 two input arguments and returning one output. The underlying function to call is passed in as *func*. The letters correspond to dtypechar's of the specific data type supported by the general-purpose function. The argument `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 **PyUFunc_O_O**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

void **PyUFunc_OO_O**(*char** args, npy_intp* dimensions, npy_intp* steps, void* 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 *func* and it must accept calls with the signature `(PyObject*)(PyObject*)` for `PyUFunc_O_O` or `(PyObject*)(PyObject *, PyObject *)` for `PyUFunc_OO_O`.

void **PyUFunc_O_O_method**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object. For each iteration of the loop, the Python obejct is extracted from the array and its *func* method is called returning the result to the output array.

void **PyUFunc_OO_O_method**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object that takes one argument. The first argument in *args* is the method whose function is called, the second argument in *args* is the argument passed to the function. The output of the function is stored in the third entry of *args*.

void **PyUFunc_On_Om**(*char** args, npy_intp* dimensions, npy_intp* steps, void* func*)

This is the 1-d core function used by the dynamic ufuncs created by umath.frompyfunc(function, nin, nout). In this case *func* is a pointer to a `PyUFunc_PyFuncData` structure which has definition

PyUFunc_PyFuncData

```
typedef struct {
    int nin;
    int nout;
    PyObject *callable;
} PyUFunc_PyFuncData;
```

At each iteration of the loop, the *nin* input objects are exctracted from their object arrays and placed into an argument tuple, the Python *callable* is called with the input arguments, and the nout outputs are placed into their object arrays.

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 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...).

5.6.1 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

This macro is defined to a positive inf. The corresponding single and extension precision macro are available with the suffix F and L.

NPY_PZERO
This macro is defined to positive zero. The corresponding single and extension precision macro are available with the suffix F and L.

NPY_NZERO
This macro is defined to negative zero (that is with the sign bit set). The corresponding single and extension precision macro are available with the suffix F and L.

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.

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.

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).

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).

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 (π)

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 ($\frac{2}{\pi}$)

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 pre-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 explaine 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-condiction 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 calcluated. 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 calcluated.

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

$$\begin{aligned} o &= & i[0] \\ o &= & i[k]\text{<op>}o \quad k = 1 \ldots N \end{aligned}$$

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

$$\begin{aligned} o[0] &= & i[0] \\ o[k] &= & i[k]\text{<op>}o[k-1] \quad k = 1 \ldots N. \end{aligned}$$

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 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 an 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.

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

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.

The Documentation Marathon is still ongoing. Please help us write better documentation for Numpy by joining it! Instructions on how to join and what to do can be found on the scipy.org website

INDEX