The N-dimensional array (`ndarray`)

An **`ndarray`** (generated/numpy.ndarray.html#numpy.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`** (generated/numpy.ndarray.shape.html#numpy.ndarray.shape), which is a **`tuple`** (https://docs.python.org/dev/library/stdtypes.html#tuple) of *N* positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (dtype) (arrays.dtypes.html#arrays-dtypes), one of which is associated with each ndarray.

As with other container objects in Python, the contents of an **`ndarray`** (generated/numpy.ndarray.html#numpy.ndarray) can be accessed and modified by indexing or slicing (arrays.indexing.html#arrays-indexing) the array (using, for example, *N* integers), and via the methods and attributes of the **`ndarray`** (generated/numpy.ndarray.html#numpy.ndarray).

Different **`ndarrays`** (generated/numpy.ndarray.html#numpy.ndarray) can share the same data, so that changes made in one **`ndarray`** (generated/numpy.ndarray.html#numpy.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`** (https://docs.python.org/dev/library/stdtypes.html#str) or objects implementing the **`buffer`** or array (arrays.interface.html#arrays-interface) 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 Python container-like syntax:

```
>>> # The element of x in the *second* row, *third* column, namely, 6.
>>> x[1, 2]
```

For example slicing (arrays.indexing.html#arrays-indexing) can produce views of the array:

```
>>> y = x[:,1]
>>> y
array([2, 5])
>>> y[0] = 9 # this also changes the corresponding element in x
>>> y
array([9, 5])
>>> x
array([[1, 9, 3],
       [4, 5, 6]])
```

Constructing arrays

New arrays can be constructed using the routines detailed in Array creation routines (routines.array-creation.html#routines-array-creation), and also by using the low-level **`ndarray`** (generated/numpy.ndarray.html#numpy.ndarray) constructor:

`ndarray` (generated/numpy.ndarray.html#numpy.ndarray)	An array object represents a multidimensional, homogeneous array of fixed-size items.

Indexing arrays

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

See also:
Array Indexing (arrays.indexing.html#arrays-indexing).

Internal memory layout of an ndarray

An instance of class **`ndarray`** (generated/numpy.ndarray.html#numpy.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`** (generated/numpy.ndarray.shape.html#numpy.ndarray.shape) of the array. How many bytes each item takes and how the bytes are interpreted is defined by the data-type object (arrays.dtypes.html#arrays-dtypes) associated with the array.

A segment of memory is inherently 1-dimensional, and there are many different schemes for arranging the items of an *N*-dimensional array in a 1-dimensional block. NumPy is flexible, and **`ndarray`** (generated/numpy.ndarray.html#numpy.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`** (generated/numpy.ndarray.strides.html#numpy.ndarray.strides) of the array. The column-major (../glossary.html#term-column-major) order (used, for example, in the Fortran language and in *Matlab*) and row-major (../glossary.html#term-row-major) order (used in C) schemes are just specific kinds of strided scheme, and correspond to memory that can be *addressed* by the strides:

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

where d_j = *self.shape[j]*.

Both the C and Fortran orders are contiguous (https://docs.python.org/dev/glossary.html#term-contiguous), *i.e.,* single-segment, memory layouts, in which every part of the memory block can be accessed by some combination of the indices.

While a C-style and Fortran-style contiguous array, which has the corresponding flags set, can be addressed with the above strides, the actual strides may be different. This can happen in two cases:

1. If `self.shape[k] == 1` then for any legal index `index[k] == 0`. This means that in the formula for the offset $n_k = 0$ and thus $s_k n_k = 0$ and the value of s_k = *self.strides[k]* is arbitrary.
2. If an array has no elements (`self.size == 0`) there is no legal index and the strides are never used. Any array with no elements may be considered C-style and Fortran-style contiguous.

Point 1. means that `self` and `self.squeeze()` always have the same contiguity and aligned flags value. This also means that even a high dimensional array could be C-style and Fortran-style contiguous at the same time.

An array is considered aligned if the memory offsets for all elements and the base offset itself is a multiple of *self.itemsize*.

Note:
Points (1) and (2) are not yet applied by default. Beginning with NumPy 1.8.0, they are applied consistently only if the environment variable `NPY_RELAXED_STRIDES_CHECKING=1` was defined when NumPy was built. Eventually this will become the default.

You can check whether this option was enabled when your NumPy was built by looking at the value of `np.ones((10,1), order='C').flags.f_contiguous`. If this is `True`, then your NumPy has relaxed strides checking enabled.

Warning:
It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Data in new **`ndarrays`** (generated/numpy.ndarray.html#numpy.ndarray) is in the row-major (../glossary.html#term-row-major) (C) order, unless otherwise specified, but, for example, basic array slicing (arrays.indexing.html#arrays-indexing) often produces views (../glossary.html#term-view) 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.

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 (generated/numpy.ndarray.flags.html#numpy.ndarray.flags)	Information about the memory layout of the array.
ndarray.shape (generated/numpy.ndarray.shape.html#numpy.ndarray.shape)	Tuple of array dimensions.
ndarray.strides (generated/numpy.ndarray.strides.html#numpy.ndarray.strides)	Tuple of bytes to step in each dimension when traversing an array.
ndarray.ndim (generated/numpy.ndarray.ndim.html#numpy.ndarray.ndim)	Number of array dimensions.
ndarray.data (generated/numpy.ndarray.data.html#numpy.ndarray.data)	Python buffer object pointing to the start of the array's data.
ndarray.size (generated/numpy.ndarray.size.html#numpy.ndarray.size)	Number of elements in the array.
ndarray.itemsize (generated/numpy.ndarray.itemsize.html#numpy.ndarray.itemsize)	Length of one array element in bytes.
ndarray.nbytes (generated/numpy.ndarray.nbytes.html#numpy.ndarray.nbytes)	Total bytes consumed by the elements of the array.
ndarray.base (generated/numpy.ndarray.base.html#numpy.ndarray.base)	Base object if memory is from some other object.

Data type

See also:
Data type objects (arrays.dtypes.html#arrays-dtypes)

The data type object associated with the array can be found in the **dtype** (generated/numpy.ndarray.dtype.html#numpy.ndarray.dtype) attribute:

ndarray.dtype (generated/numpy.ndarray.dtype.html#numpy.ndarray.dtype)	Data-type of the array's elements.

Other attributes

ndarray.T (generated/numpy.ndarray.T.html#numpy.ndarray.T)	Same as self.transpose(), except that self is returned if self.ndim < 2.
ndarray.real (generated/numpy.ndarray.real.html#numpy.ndarray.real)	The real part of the array.
ndarray.imag (generated/numpy.ndarray.imag.html#numpy.ndarray.imag)	The imaginary part of the array.
ndarray.flat (generated/numpy.ndarray.flat.html#numpy.ndarray.flat)	A 1-D iterator over the array.
ndarray.ctypes (generated/numpy.ndarray.ctypes.html#numpy.ndarray.ctypes)	An object to simplify the interaction of the array with the ctypes module.

Array interface

See also:
The Array Interface (arrays.interface.html#arrays-interface).

`__array_interface__` (arrays.interface.html#__array_interface__)	Python-side of the array interface
`__array_struct__`	C-side of the array interface

`ctypes` (https://docs.python.org/dev/library/ctypes.html#module-ctypes) foreign function interface

`ndarray.ctypes` (generated/numpy.ndarray.ctypes.html#numpy.ndarray.ctypes)	An object to simplify the interaction of the array with the ctypes module.

Array methods

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

For the following methods there are also corresponding functions in `numpy` (index.html#module-numpy): `all` (generated/numpy.all.html#numpy.all), `any` (generated/numpy.any.html#numpy.any), `argmax` (generated/numpy.argmax.html#numpy.argmax), `argmin` (generated/numpy.argmin.html#numpy.argmin), `argpartition` (generated/numpy.argpartition.html#numpy.argpartition), `argsort` (generated/numpy.argsort.html#numpy.argsort), `choose` (generated/numpy.choose.html#numpy.choose), `clip` (generated/numpy.clip.html#numpy.clip), `compress` (generated/numpy.compress.html#numpy.compress), `copy` (generated/numpy.copy.html#numpy.copy), `cumprod` (generated/numpy.cumprod.html#numpy.cumprod), `cumsum` (generated/numpy.cumsum.html#numpy.cumsum), `diagonal` (generated/numpy.diagonal.html#numpy.diagonal), `imag` (generated/numpy.imag.html#numpy.imag), `max` (generated/numpy.amax.html#numpy.amax), `mean` (generated/numpy.mean.html#numpy.mean), `min` (generated/numpy.amin.html#numpy.amin), `nonzero` (generated/numpy.nonzero.html#numpy.nonzero), `partition` (generated/numpy.partition.html#numpy.partition), `prod` (generated/numpy.prod.html#numpy.prod), `ptp` (generated/numpy.ptp.html#numpy.ptp), `put` (generated/numpy.put.html#numpy.put), `ravel` (generated/numpy.ravel.html#numpy.ravel), `real` (generated/numpy.real.html#numpy.real), `repeat` (generated/numpy.repeat.html#numpy.repeat), `reshape` (generated/numpy.reshape.html#numpy.reshape), `round` (generated/numpy.around.html#numpy.around), `searchsorted` (generated/numpy.searchsorted.html#numpy.searchsorted), `sort` (generated/numpy.sort.html#numpy.sort), `squeeze` (generated/numpy.squeeze.html#numpy.squeeze), `std` (generated/numpy.std.html#numpy.std), `sum` (generated/numpy.sum.html#numpy.sum), `swapaxes` (generated/numpy.swapaxes.html#numpy.swapaxes), `take` (generated/numpy.take.html#numpy.take), `trace` (generated/numpy.trace.html#numpy.trace), `transpose` (generated/numpy.transpose.html#numpy.transpose), `var` (generated/numpy.var.html#numpy.var).

Array conversion

`ndarray.item` (generated/numpy.ndarray.item.html#numpy.ndarray.item)(*args)	Copy an element of an array to a standard Python scalar and return it.
`ndarray.tolist` (generated/numpy.ndarray.tolist.html#numpy.ndarray.tolist)()	Return the array as a (possibly nested) list.
`ndarray.itemset` (generated/numpy.ndarray.itemset.html#numpy.ndarray.itemset)	Insert scalar into

(*args)	an array (scalar is cast to array's dtype, if possible)
`ndarray.tostring` (generated/numpy.ndarray.tostring.html#numpy.ndarray.tostring) ([order])	Construct Python bytes containing the raw data bytes in the array.
`ndarray.tobytes` (generated/numpy.ndarray.tobytes.html#numpy.ndarray.tobytes) ([order])	Construct Python bytes containing the raw data bytes in the array.
`ndarray.tofile` (generated/numpy.ndarray.tofile.html#numpy.ndarray.tofile) (fid[, sep, format])	Write array to a file as text or binary (default).
`ndarray.dump` (generated/numpy.ndarray.dump.html#numpy.ndarray.dump)(file)	Dump a pickle of the array to the specified file.
`ndarray.dumps` (generated/numpy.ndarray.dumps.html#numpy.ndarray.dumps)()	Returns the pickle of the array as a string.
`ndarray.astype` (generated/numpy.ndarray.astype.html#numpy.ndarray.astype) (dtype[, order, casting, ...])	Copy of the array, cast to a specified type.
`ndarray.byteswap` (generated/numpy.ndarray.byteswap.html#numpy.ndarray.byteswap)(inplace)	Swap the bytes of the array elements
`ndarray.copy` (generated/numpy.ndarray.copy.html#numpy.ndarray.copy)([order])	Return a copy of the array.
`ndarray.view` (generated/numpy.ndarray.view.html#numpy.ndarray.view) ([dtype, type])	New view of array with the same data.
`ndarray.getfield` (generated/numpy.ndarray.getfield.html#numpy.ndarray.getfield) (dtype[, offset])	Returns a field of the given array as a certain type.
`ndarray.setflags` (generated/numpy.ndarray.setflags.html#numpy.ndarray.setflags) ([write, align, uic])	Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
`ndarray.fill` (generated/numpy.ndarray.fill.html#numpy.ndarray.fill)(value)	Fill the array with a scalar value.

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` (generated/numpy.ndarray.reshape.html#numpy.ndarray.reshape)(shape[, order])	Returns an array containing the same data with a new

shape.	
ndarray.resize (generated/numpy.ndarray.resize.html#numpy.ndarray.resize)(new_shape[, refcheck])	Change shape and size of array in-place.
ndarray.transpose(generated/numpy.ndarray.transpose.html#numpy.ndarray.transpose)(*axes)	Returns a view of the array with axes transposed.
ndarray.swapaxes(generated/numpy.ndarray.swapaxes.html#numpy.ndarray.swapaxes)(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
ndarray.flatten (generated/numpy.ndarray.flatten.html#numpy.ndarray.flatten)([order])	Return a copy of the array collapsed into one dimension.
ndarray.ravel (generated/numpy.ndarray.ravel.html#numpy.ndarray.ravel)([order])	Return a flattened array.
ndarray.squeeze(generated/numpy.ndarray.squeeze.html#numpy.ndarray.squeeze)([axis])	Remove single-dimensional entries from the shape of *a*.

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 (generated/numpy.ndarray.take.html#numpy.ndarray.take)(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
ndarray.put (generated/numpy.ndarray.put.html#numpy.ndarray.put)(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ndarray.repeat (generated/numpy.ndarray.repeat.html#numpy.ndarray.repeat)(repeats[, axis])	Repeat elements of an array.
ndarray.choose (generated/numpy.ndarray.choose.html#numpy.ndarray.choose)(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
ndarray.sort (generated/numpy.ndarray.sort.html#numpy.ndarray.sort)([axis, kind, order])	Sort an array, in-place.
ndarray.argsort (generated/numpy.ndarray.argsort.html#numpy.ndarray.argsort)([axis, kind, order])	Returns the indices that would sort this array.
ndarray.partition (generated/numpy.ndarray.partition.html#numpy.ndarray.partition)(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array.
ndarray.argpartition(generated/numpy.ndarray.argpartition.html#numpy.ndarray.argpartition)(kth[, axis, kind, order])	Returns the indices that would partition this array.
ndarray.searchsorted	Find indices where

(generated/numpy.ndarray.searchsorted.html#numpy.ndarray.searchsorted)(v[, side, sorter])	elements of v should be inserted in a to maintain order.
`ndarray.nonzero` (generated/numpy.ndarray.nonzero.html#numpy.ndarray.nonzero)()	Return the indices of the elements that are non-zero.
`ndarray.compress` (generated/numpy.ndarray.compress.html#numpy.ndarray.compress) (condition[, axis, out])	Return selected slices of this array along given axis.
`ndarray.diagonal` (generated/numpy.ndarray.diagonal.html#numpy.ndarray.diagonal) ([offset, axis1, axis2])	Return specified diagonals.

Calculation

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

- If *axis* is *None* (the default), the array is treated as a 1-D array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. (An array scalar is an instance of the types/classes float32, float64, etc., whereas a 0-dimensional array is an ndarray instance containing precisely one array scalar.)
- If *axis* is an integer, then the operation is done over the given axis (for each 1-D subarray that can be created along the given axis).

Example of the *axis* argument:
A 3-dimensional array of size 3 x 3 x 3, summed over each of its three axes

```
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],
       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],
       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> x.sum(axis=0)
array([[27, 30, 33],
       [36, 39, 42],
       [45, 48, 51]])
>>> # for sum, axis is the first keyword, so we may omit it,
>>> # specifying only its value
>>> x.sum(0), x.sum(1), x.sum(2)
(array([[27, 30, 33],
        [36, 39, 42],
        [45, 48, 51]]),
 array([[ 9, 12, 15],
        [36, 39, 42],
        [63, 66, 69]]),
 array([[ 3, 12, 21],
        [30, 39, 48],
        [57, 66, 75]]))
```

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`** (generated/numpy.ndarray.html#numpy.ndarray) and have the same number of elements. It can have a different data type in which case casting will be performed.

`ndarray.argmax` (generated/numpy.ndarray.argmax.html#numpy.ndarray.argmax)([axis, out])	Return indices of the maximum values along the given axis.
`ndarray.min` (generated/numpy.ndarray.min.html#numpy.ndarray.min)([axis, out, keepdims])	Return the minimum along a given axis.
`ndarray.argmin` (generated/numpy.ndarray.argmin.html#numpy.ndarray.argmin)([axis, out])	Return indices of the minimum values along the given axis of *a*.
`ndarray.ptp` (generated/numpy.ndarray.ptp.html#numpy.ndarray.ptp)([axis, out])	Peak to peak (maximum - minimum) value along a given axis.
`ndarray.clip` (generated/numpy.ndarray.clip.html#numpy.ndarray.clip)([min, max, out])	Return an array whose values are limited to `[min, max]`.
`ndarray.conj` (generated/numpy.ndarray.conj.html#numpy.ndarray.conj)()	Complex-conjugate all elements.
`ndarray.round` (generated/numpy.ndarray.round.html#numpy.ndarray.round)([decimals, out])	Return *a* with each element rounded to the given number of decimals.
`ndarray.trace` (generated/numpy.ndarray.trace.html#numpy.ndarray.trace)([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
`ndarray.sum` (generated/numpy.ndarray.sum.html#numpy.ndarray.sum)([axis, dtype, out, keepdims])	Return the sum of the array elements over the given axis.

ndarray.cumsum (generated/numpy.ndarray.cumsum.html#numpy.ndarray.cumsum)([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
ndarray.mean (generated/numpy.ndarray.mean.html#numpy.ndarray.mean)([axis, dtype, out, keepdims])	Returns the average of the array elements along given axis.
ndarray.var (generated/numpy.ndarray.var.html#numpy.ndarray.var)([axis, dtype, out, ddof, keepdims])	Returns the variance of the array elements, along given axis.
ndarray.std (generated/numpy.ndarray.std.html#numpy.ndarray.std)([axis, dtype, out, ddof, keepdims])	Returns the standard deviation of the array elements along given axis.
ndarray.prod (generated/numpy.ndarray.prod.html#numpy.ndarray.prod)([axis, dtype, out, keepdims])	Return the product of the array elements over the given axis
ndarray.cumprod (generated/numpy.ndarray.cumprod.html#numpy.ndarray.cumprod)([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
ndarray.all (generated/numpy.ndarray.all.html#numpy.ndarray.all)([axis, out, keepdims])	Returns True if all elements evaluate to True.
ndarray.any (generated/numpy.ndarray.any.html#numpy.ndarray.any)([axis, out, keepdims])	Returns True if any of the elements of *a* evaluate to True.

Arithmetic, matrix multiplication, and comparison operations

Arithmetic and comparison operations on **`ndarrays`** (generated/numpy.ndarray.html#numpy.ndarray) are defined as element-wise operations, and generally yield **`ndarray`** (generated/numpy.ndarray.html#numpy.ndarray) objects as results.

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

Comparison operators:

`ndarray.__lt__` (generated/numpy.ndarray.__lt__.html#numpy.ndarray.__lt__) x.__lt__(y) <==> x<y

`ndarray.__le__` (generated/numpy.ndarray.__le__.html#numpy.ndarray.__le__) x.__le__(y) <==> x<=y

`ndarray.__gt__` (generated/numpy.ndarray.__gt__.html#numpy.ndarray.__gt__) x.__gt__(y) <==> x>y

`ndarray.__ge__` (generated/numpy.ndarray.__ge__.html#numpy.ndarray.__ge__) x.__ge__(y) <==> x>=y

`ndarray.__eq__` (generated/numpy.ndarray.__eq__.html#numpy.ndarray.__eq__) x.__eq__(y) <==> x==y

`ndarray.__ne__` (generated/numpy.ndarray.__ne__.html#numpy.ndarray.__ne__) x.__ne__(y) <==> x!=y

Truth value of an array (**`bool`**):

`ndarray.__nonzero__` (generated/numpy.ndarray.__nonzero__.html#numpy.ndarray.__nonzero__) x.__nonzero__() <==> x != 0

Note:
Truth-value testing of an array invokes **`ndarray.__nonzero__`** (generated/numpy.ndarray.__nonzero__.html#numpy.ndarray.__nonzero__), which raises an error if the number of elements in the array is larger than 1, because the truth value of such arrays is ambiguous. Use **`.any()`** (generated/numpy.ndarray.any.html#numpy.ndarray.any) and **`.all()`** (generated/numpy.ndarray.all.html#numpy.ndarray.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__` (generated/numpy.ndarray.__neg__.html#numpy.ndarray.__neg__) x.__neg__() <==> -x

`ndarray.__pos__` (generated/numpy.ndarray.__pos__.html#numpy.ndarray.__pos__) x.__pos__() <==> +x

`ndarray.__abs__` (generated/numpy.ndarray.__abs__.html#numpy.ndarray.__abs__)() <==> abs(x)

`ndarray.__invert__` (generated/numpy.ndarray.__invert__.html#numpy.ndarray.__invert__) x.__invert__() <==> ~x

Arithmetic:

`ndarray.__add__` (generated/numpy.ndarray.__add__.html#numpy.ndarray.__add__) x.__add__(y) <==> x+y

`ndarray.__sub__` (generated/numpy.ndarray.__sub__.html#numpy.ndarray.__sub__) x.__sub__(y) <==> x-y

`ndarray.__mul__` (generated/numpy.ndarray.__mul__.html#numpy.ndarray.__mul__) x.__mul__(y)

	<==> x*y
ndarray.__div__ (generated/numpy.ndarray.__div__.html#numpy.ndarray.__div__)	x.__div__(y) <==> x/y
ndarray.__truediv__ (generated/numpy.ndarray.__truediv__.html#numpy.ndarray.__truediv__)	x.__truediv__(y) <==> x/y
ndarray.__floordiv__ (generated/numpy.ndarray.__floordiv__.html#numpy.ndarray.__floordiv__)	x.__floordiv__(y) <==> x//y
ndarray.__mod__ (generated/numpy.ndarray.__mod__.html#numpy.ndarray.__mod__)	x.__mod__(y) <==> x%y
ndarray.__divmod__ (generated/numpy.ndarray.__divmod__.html#numpy.ndarray.__divmod__)(y) <==> divmod(x, y)	
ndarray.__pow__ (generated/numpy.ndarray.__pow__.html#numpy.ndarray.__pow__) (y[, z]) <==> pow(x, y[, z])	
ndarray.__lshift__ (generated/numpy.ndarray.__lshift__.html#numpy.ndarray.__lshift__)	x.__lshift__(y) <==> x<<y
ndarray.__rshift__ (generated/numpy.ndarray.__rshift__.html#numpy.ndarray.__rshift__)	x.__rshift__(y) <==> x>>y
ndarray.__and__ (generated/numpy.ndarray.__and__.html#numpy.ndarray.__and__)	x.__and__(y) <==> x&y
ndarray.__or__ (generated/numpy.ndarray.__or__.html#numpy.ndarray.__or__)	x.__or__(y) <==> xly
ndarray.__xor__ (generated/numpy.ndarray.__xor__.html#numpy.ndarray.__xor__)	x.__xor__(y) <==> x^y

Note:

- Any third argument to **pow** (https://docs.python.org/dev/library/functions.html#pow) is silently ignored, as the underlying **ufunc** (generated/numpy.power.html#numpy.power) takes only two arguments.
- The three division operators are all defined; **div** is active by default, **truediv** is active when **__future__** (https://docs.python.org/dev/library/__future__.html#module-__future__) division is in effect.
- Because **ndarray** (generated/numpy.ndarray.html#numpy.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__ (generated/numpy.ndarray.__iadd__.html#numpy.ndarray.__iadd__)	x.__iadd__(y) <==> x+=y
ndarray.__isub__ (generated/numpy.ndarray.__isub__.html#numpy.ndarray.__isub__)	x.__isub__(y) <==> x-=y
ndarray.__imul__ (generated/numpy.ndarray.__imul__.html#numpy.ndarray.__imul__)	x.__imul__(y) <==> x*=y
ndarray.__idiv__ (generated/numpy.ndarray.__idiv__.html#numpy.ndarray.__idiv__)	x.__idiv__(y) <==> x/=y
ndarray.__itruediv__ (generated/numpy.ndarray.__itruediv__.html#numpy.ndarray.__itruediv__)	x.__itruediv__(y) <==> x/y
ndarray.__ifloordiv__ (generated/numpy.ndarray.__ifloordiv__.html#numpy.ndarray.__ifloordiv__)	x.__ifloordiv__(y) <==> x//y
ndarray.__imod__	x.__imod__(y) <==> x%=y

(generated/numpy.ndarray.__imod__.html#numpy.ndarray.__imod__)

ndarray.__ipow__ (generated/numpy.ndarray.__ipow__.html#numpy.ndarray.__ipow__)	x.__ipow__(y) <==> x**=y
ndarray.__ilshift__ (generated/numpy.ndarray.__ilshift__.html#numpy.ndarray.__ilshift__)	x.__ilshift__(y) <==> x<<=y
ndarray.__irshift__ (generated/numpy.ndarray.__irshift__.html#numpy.ndarray.__irshift__)	x.__irshift__(y) <==> x>>=y
ndarray.__iand__ (generated/numpy.ndarray.__iand__.html#numpy.ndarray.__iand__)	x.__iand__(y) <==> x&=y
ndarray.__ior__ (generated/numpy.ndarray.__ior__.html#numpy.ndarray.__ior__)	x.__ior__(y) <==> x|=y
ndarray.__ixor__ (generated/numpy.ndarray.__ixor__.html#numpy.ndarray.__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.

Matrix Multiplication:

ndarray.__matmul__

Note:
Matrix operators `@` and `@=` were introduced in Python 3.5 following PEP465. NumPy 1.10.0 has a preliminary implementation of `@` for testing purposes. Further documentation can be found in the **matmul** (generated/numpy.matmul.html#numpy.matmul) documentation.

Special methods

For standard library functions:

ndarray.__copy__ (generated/numpy.ndarray.__copy__.html#numpy.ndarray.__copy__)([order])	Return a copy of the array.
ndarray.__deepcopy__ (generated/numpy.ndarray.__deepcopy__.html#numpy.ndarray.__deepcopy__)(() -> Deep copy of array.)	Used if copy.deepcopy is called on an array.
ndarray.__reduce__ (generated/numpy.ndarray.__reduce__.html#numpy.ndarray.__reduce__)()	For pickling.
ndarray.__setstate__ (generated/numpy.ndarray.__setstate__.html#numpy.ndarray.__setstate__)(version, shape, dtype, ...)	For unpickling.

Basic customization:

ndarray.__new__ (generated/numpy.ndarray.__new__.html#numpy.ndarray.__new__)((S, ...)	
ndarray.__array__ (generated/numpy.ndarray.__array__.html#numpy.ndarray.__array__)(...)	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.

`ndarray.__array_wrap__`
(generated/numpy.ndarray.__array_wrap__.html#numpy.ndarray.__array_wrap__)(...)

Container customization: (see Indexing (arrays.indexing.html#arrays-indexing))

`ndarray.__len__`
(generated/numpy.ndarray.__len__.html#numpy.ndarray.__len__)() <==> len(x)

`ndarray.__getitem__`	x.__getitem__(y) <==>
(generated/numpy.ndarray.__getitem__.html#numpy.ndarray.__getitem__)	x[y]
`ndarray.__setitem__`	x.__setitem__(i, y)
(generated/numpy.ndarray.__setitem__.html#numpy.ndarray.__setitem__)	<==> x[i]=y
`ndarray.__contains__`	x.__contains__(y)
(generated/numpy.ndarray.__contains__.html#numpy.ndarray.__contains__)	<==> y in x

Conversion; the operations `complex`, `int`, `long`, `float`, `oct` (https://docs.python.org/dev/library/functions.html#oct), and `hex` (https://docs.python.org/dev/library/functions.html#hex). They work only on arrays that have one element in them and return the appropriate scalar.

`ndarray.__int__`
(generated/numpy.ndarray.__int__.html#numpy.ndarray.__int__)() <==> int(x)
`ndarray.__long__`
(generated/numpy.ndarray.__long__.html#numpy.ndarray.__long__)() <==> long(x)
`ndarray.__float__`
(generated/numpy.ndarray.__float__.html#numpy.ndarray.__float__)() <==> float(x)
`ndarray.__oct__`
(generated/numpy.ndarray.__oct__.html#numpy.ndarray.__oct__)() <==> oct(x)
`ndarray.__hex__`
(generated/numpy.ndarray.__hex__.html#numpy.ndarray.__hex__)() <==> hex(x)

String representations:

`ndarray.__str__`
(generated/numpy.ndarray.__str__.html#numpy.ndarray.__str__)() <==> str(x)
`ndarray.__repr__`
(generated/numpy.ndarray.__repr__.html#numpy.ndarray.__repr__)() <==> repr(x)

Table Of Contents (../contents.html)

        - `ctypes` foreign function interface
    - Array methods
        - Array conversion
        - Shape manipulation
        - Item selection and manipulation
        - Calculation
    - Arithmetic, matrix multiplication, and comparison operations
    - Special methods

Previous topic

Array objects (arrays.html)

Next topic

numpy.ndarray (generated/numpy.ndarray.html)