

This is an archival dump of old wiki content --- see [scipy.org](https://docs.scipy.org/doc/numpy-dev/user/quickstart.html) for current material.
Please see <https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>

Titles

Text

- [Login](#)

SciPy

- [Tentative_NumPy_Tutorial](#)
- [RecentChanges](#)
- [FindPage](#)
- [HelpContents](#)
- [Tentative_NumPy_Tutorial](#)
- [Attachments](#)
- More Actions:

Please do not hesitate to click the *edit* button. You will need to create a [User Account](#) first.

Contents

1. [Prerequisites](#)
2. [The Basics](#)
 1. [An example](#)
 2. [Array Creation](#)
 3. [Printing Arrays](#)
 4. [Basic Operations](#)
 5. [Universal Functions](#)
 6. [Indexing, Slicing and Iterating](#)
3. [Shape Manipulation](#)
 1. [Changing the shape of an array](#)
 2. [Stacking together different arrays](#)
 3. [Splitting one array into several smaller ones](#)
4. [Copies and Views](#)
 1. [No Copy at All](#)
 2. [View or Shallow Copy](#)
 3. [Deep Copy](#)
 4. [Functions and Methods Overview](#)
5. [Less Basic](#)
 1. [Broadcasting rules](#)
6. [Fancy indexing and index tricks](#)
 1. [Indexing with Arrays of Indices](#)
 2. [Indexing with Boolean Arrays](#)
 3. [The ix_\(\) function](#)
 4. [Indexing with strings](#)
7. [Linear Algebra](#)
 1. [Simple Array Operations](#)
 2. [The Matrix Class](#)
 3. [Indexing: Comparing Matrices and 2D Arrays](#)
8. [Tricks and Tips](#)
 1. ["Automatic" Reshaping](#)
 2. [Vector Stacking](#)
 3. [Histograms](#)
9. [References](#)

Prerequisites

Before reading this tutorial you should know a bit of Python. If you would like to refresh your memory, take a look at the [Python tutorial](#).

If you wish to work the examples in this tutorial, you must also have some software installed on your computer. Minimally:

- [Python](#)
- [NumPy](#)

These you may find useful:

- [ipython](#) is an enhanced interactive Python shell which is very convenient for exploring NumPy's features
- [matplotlib](#) will enable you to plot graphics

- [SciPy](#) provides a lot of scientific routines that work on top of NumPy

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy dimensions are called *axes*. The number of axes is *rank*.

For example, the coordinates of a point in 3D space `[1, 2, 1]` is an array of rank 1, because it has one axis. That axis has a length of 3. In example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy's array class is called `ndarray`. It is also known by the alias `array`. Note that `numpy.array` is not the same as the Standard Python Library class `array.array`, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an `ndarray` object are:

`ndarray.ndim`

the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as *rank*.

`ndarray.shape`

the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, `shape` will be `(n,m)`. The length of the `shape` tuple is therefore the rank, or number of dimensions, `ndim`.

`ndarray.size`

the total number of elements of the array. This is equal to the product of the elements of `shape`.

`ndarray.dtype`

an object describing the type of the elements in the array. One can create or specify `dtype`'s using standard Python types. Additionally NumPy provides types of its own. `numpy.int32`, `numpy.int16`, and `numpy.float64` are some examples.

`ndarray.itemsize`

the size in bytes of each element of the array. For example, an array of elements of type `float64` has `itemsize` 8 (=64/8), while one of type `complex32` has `itemsize` 4 (=32/8). It is equivalent to `ndarray.dtype.itemsize`.

`ndarray.data`

the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

An example

```
>>> from numpy import *
>>> a = arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
>>> a.itemsize
4
>>> a.size
15
>>> type(a)
numpy.ndarray
>>> b = array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
numpy.ndarray
```

Array Creation

There are several ways to create arrays.

For example, you can create an array from a regular Python list or tuple using the `array` function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> from numpy import *
>>> a = array( [2,3,4] )
>>> a
array([2, 3, 4])
>>> a.dtype
```

```
dtype('int32')
>>> b = array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

A frequent error consists in calling `array` with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = array(1,2,3,4)      # WRONG
>>> a = array([1,2,3,4])   # RIGHT
```

`array` transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = array( [ (1.5,2,3), (4,5,6) ] )
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, [NumPy](#) offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function `zeros` creates an array full of zeros, the function `ones` creates an array full of ones, and the function `empty` creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is `float64`.

```
>>> zeros( (3,4) )
array([[0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.]])
>>> ones( (2,3,4), dtype=int16 )           # dtype can also be specified
array([[[ 1,  1,  1,  1],
        [ 1,  1,  1,  1],
        [ 1,  1,  1,  1]],
       [[ 1,  1,  1,  1],
        [ 1,  1,  1,  1],
        [ 1,  1,  1,  1]]], dtype=int16)
>>> empty( (2,3) )
array([[ 3.73603959e-262,  6.02658058e-154,  6.55490914e-260],
       [ 5.30498948e-313,  3.14673309e-307,  1.00000000e+000]])
```

To create sequences of numbers, NumPy provides a function analogous to `range` that returns arrays instead of lists

```
>>> arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> arange( 0, 2, 0.3 )           # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When `arange` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function `linspace` that receives as an argument the number of elements that we want, instead of the step:

```
>>> linspace( 0, 2, 9 )           # 9 numbers from 0 to 2
array([ 0. ,  0.25,  0.5 ,  0.75,  1. ,  1.25,  1.5 ,  1.75,  2. ])
>>> x = linspace( 0, 2*pi, 100 )  # useful to evaluate function at lots of points
>>> f = sin(x)
```

See also

[array](#), [zeros](#), [zeros_like](#), [ones](#), [ones_like](#), [empty](#), [empty_like](#), [arange](#), [linspace](#), [rand](#), [randn](#), [fromfunction](#), [fromfile](#)

Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
>>> a = arange(6)                  # 1d array
>>> print a
[0 1 2 3 4 5]
>>>
>>> b = arange(12).reshape(4,3)   # 2d array
>>> print b
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
```

```
>>> c = arange(24).reshape(2,3,4)          # 3d array
>>> print c
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

See [below](#) to get more details on reshape.

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>> print arange(10000)
[  0    1    2 ..., 9997 9998 9999]
>>>
>>> print arange(10000).reshape(100,100)
[[  0    1    2 ...,  97  98  99]
 [100 101 102 ..., 197 198 199]
 [200 201 202 ..., 297 298 299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using `set_printoptions`.

```
>>> set_printoptions(threshold='nan')
```

Basic Operations

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = array( [20,30,40,50] )
>>> b = arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True,  True, False, False], dtype=bool)
```

Unlike in many matrix languages, the product operator `*` operates elementwise in NumPy arrays. The matrix product can be performed using the `dot` function or creating matrix objects (see matrix section of this tutorial).

```
>>> A = array( [[1,1],
...            [0,1]] )
>>> B = array( [[2,0],
...            [3,4]] )
>>> A*B                                # elementwise product
array([[2, 0],
       [0, 4]])
>>> dot(A,B)                           # matrix product
array([[5, 4],
       [3, 4]])
```

Some operations, such as `+=` and `*=`, act in place to modify an existing array rather than create a new one.

```
>>> a = ones((2,3), dtype=int)
>>> b = random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.69092703,  3.8324276 ,  3.0114541 ],
       [ 3.18679111,  3.3039349 ,  3.37600289]])
>>> a += b                                # b is converted to integer type
>>> a
array([[6, 6, 6],
       [6, 6, 6]])
```

When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as *upcasting*).

```
>>> a = ones(3, dtype=int32)
>>> b = linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([ 1.          ,  2.57079633,  4.14159265])
>>> c.dtype.name
```

```
'float64'
>>> d = exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the `ndarray` class.

```
>>> a = random.random((2,3))
>>> a
array([[ 0.6903007 ,  0.39168346,  0.16524769],
       [ 0.48819875,  0.77188505,  0.94792155]])
>>> a.sum()
3.4552372100521485
>>> a.min()
0.16524768654743593
>>> a.max()
0.9479215542670073
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the `axis` parameter you can apply an operation along the specified axis of an array:

```
>>> b = arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                                # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                                # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                             # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Universal Functions

NumPy provides familiar mathematical functions such as `sin`, `cos`, and `exp`. In [NumPy](#), these are called "universal functions"(`ufunc`). Within [NumPy](#), these functions operate elementwise on an array, producing an array as output.

```
>>> B = arange(3)
>>> B
array([0, 1, 2])
>>> exp(B)
array([ 1.          ,  2.71828183,  7.3890561 ])
>>> sqrt(B)
array([ 0.          ,  1.          ,  1.41421356])
>>> C = array([2., -1., 4.])
>>> add(B, C)
array([ 2.,  0.,  6.])
```

See also

[all](#), [alltrue](#), [any](#), [apply along axis](#), [argmax](#), [argmin](#), [argsort](#), [average](#), [bincount](#), [ceil](#), [clip](#), [conj](#), [conjugate](#), [corrcoef](#), [cov](#), [cross](#), [cumprod](#), [cumsum](#), [diff](#), [dot](#), [floor](#), [inner](#), [inv](#), [lexsort](#), [max](#), [maximum](#), [mean](#), [median](#), [min](#), [minimum](#), [nonzero](#), [outer](#), [prod](#), [re](#), [round](#), [sometrue](#), [sort](#), [std](#), [sum](#), [trace](#), [transpose](#), [var](#), [vdot](#), [vectorize](#), [where](#)

Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like [lists](#) and other Python sequences.

```
>>> a = arange(10)**3
>>> a
array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set every 2nd element to -1000
>>> a
array([-1000,  1, -1000,  27, -1000,  125,  216,  343,  512,  729])
>>> a[: :-1]          # reversed a
array([ 729,  512,  343,  216,  125, -1000,  27, -1000,  1, -1000])
>>> for i in a:
...     print i**(1/3.),
...
nan 1.0 nan 3.0 nan 5.0 6.0 7.0 8.0 9.0
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
>>> def f(x,y):
...     return 10*x+y
```

```

...
>>> b = fromfunction(f, (5,4), dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[:,1]                   # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, :]                # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])

```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices :

```

>>> b[-1]                    # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])

```

The expression within brackets in `b[i]` is treated as an `i` followed by as many instances of `:` as needed to represent the remaining axes. NumPy also allows you to write this using dots as `b[i,...]`.

The **dots** (`...`) represent as many colons as needed to produce a complete indexing tuple. For example, if `x` is a rank 5 array (i.e., it has 5 axes), then

- `x[1,2,...]` is equivalent to `x[1,2,:,:,:]`,
- `x[...,3]` to `x[:, :, :, 3]` and
- `x[4,...,5,:]` to `x[4, :, :, 5, :]`.

```

>>> c = array( [ [ [ 0,  1,  2],                # a 3D array (two stacked 2D arrays)
...             [ 10, 12, 13]],
...           [[100,101,102],
...            [110,112,113]] ] )
>>> c.shape
(2, 2, 3)
>>> c[1,...]                  # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                  # same as c[:, :, 2]
array([[ 2,  13],
       [102, 113]])

```

Iterating over multidimensional arrays is done with respect to the first axis:

```

>>> for row in b:
...     print row
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]

```

However, if one wants to perform an operation on each element in the array, one can use the `flat` attribute which is an [iterator](#) over all the elements of the array:

```

>>> for element in b.flat:
...     print element,
...
0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43

```

See also

[\[\]](#), [...](#), [newaxis](#), [ndenumerate](#), [indices](#), [index exp](#)

Shape Manipulation

Changing the shape of an array

An array has a shape given by the number of elements along each axis:

```

>>> a = floor(10*random.random((3,4)))
>>> a
array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])
>>> a.shape
(3, 4)

```

The shape of an array can be changed with various commands:

```
>>> a.ravel() # flatten the array
array([ 7.,  5.,  9.,  3.,  7.,  2.,  7.,  8.,  6.,  8.,  3.,  2.])
>>> a.shape = (6, 2)
>>> a.transpose()
array([[ 7.,  9.,  7.,  7.,  6.,  3.],
       [ 5.,  3.,  2.,  8.,  8.,  2.]])
```

The order of the elements in the array resulting from `ravel()` is normally "C-style", that is, the rightmost index "changes the fastest", so the element after `a[0,0]` is `a[0,1]`. If the array is reshaped to some other shape, again the array is treated as "C-style". Numpy normally creates arrays stored in this order, so `ravel()` will usually not need to copy its argument, but if the array was made by taking slices of another array or created with unusual options, it may need to be copied. The functions `ravel()` and `reshape()` can also be instructed, using an optional argument, to use FORTRAN-style arrays, in which the leftmost index changes the fastest.

The [reshape](#) function returns its argument with a modified shape, whereas the [resize](#) method modifies the array itself:

```
>>> a
array([[ 7.,  5.],
       [ 9.,  3.],
       [ 7.,  2.],
       [ 7.,  8.],
       [ 6.,  8.],
       [ 3.,  2.]])
>>> a.resize((2,6))
>>> a
array([[ 7.,  5.,  9.,  3.,  7.,  2.],
       [ 7.,  8.,  6.,  8.,  3.,  2.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
>>> a.reshape(3,-1)
array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])
```

See also: [shape example](#), [reshape example](#), [resize example](#), [ravel example](#)

Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = floor(10*random.random((2,2)))
>>> a
array([[ 1.,  1.],
       [ 5.,  8.]])
>>> b = floor(10*random.random((2,2)))
>>> b
array([[ 3.,  3.],
       [ 6.,  0.]])
>>> vstack((a,b))
array([[ 1.,  1.],
       [ 5.,  8.],
       [ 3.,  3.],
       [ 6.,  0.]])
>>> hstack((a,b))
array([[ 1.,  1.,  3.,  3.],
       [ 5.,  8.,  6.,  0.]])
```

The function [column_stack](#) stacks 1D arrays as columns into a 2D array. It is equivalent to [vstack](#) only for 1D arrays:

```
>>> column_stack((a,b)) # With 2D arrays
array([[ 1.,  1.,  3.,  3.],
       [ 5.,  8.,  6.,  0.]])
>>> a=array([4.,2.])
>>> b=array([2.,8.])
>>> a[:,newaxis] # This allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  2.],
       [ 2.,  8.]])
>>> vstack((a[:,newaxis],b[:,newaxis])) # The behavior of vstack is different
array([[ 4.],
       [ 2.],
       [ 2.],
       [ 8.]])
```

The function [row_stack](#), on the other hand, stacks 1D arrays as rows into a 2D array.

For arrays of with more than two dimensions, [hstack](#) stacks along their second axes, [vstack](#) stacks along their first axes, and [concatenate](#) allows for an optional arguments giving the number of the axis along which the concatenation should happen.

Note

In complex cases, [r_\[\]](#) and [c_\[\]](#) are useful for creating arrays by stacking numbers along one axis. They allow the use of range literals (":") :

```
>>> r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

When used with arrays as arguments, `r_[]` and `c_[]` are similar to `vstack` and `hstack` in their default behavior, but allow for an optional argument giving the number of the axis along which to concatenate.

See also: [hstack example](#), [vstack example](#), [column_stack example](#), [row_stack example](#), [concatenate example](#), [c_ example](#), [r_ example](#)

Splitting one array into several smaller ones

Using [hsplit](#), you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

```
>>> a = floor(10*random.random((2,12)))
>>> a
array([[ 8.,  8.,  3.,  9.,  0.,  4.,  3.,  0.,  0.,  6.,  4.,  4.],
       [ 0.,  3.,  2.,  9.,  6.,  0.,  4.,  5.,  7.,  5.,  1.,  4.]])
>>> hsplit(a,3) # Split a into 3
[array([[ 8.,  8.,  3.,  9.],
        [ 0.,  3.,  2.,  9.]]), array([[ 0.,  4.,  3.,  0.],
        [ 6.,  0.,  4.,  5.]]), array([[ 0.,  6.,  4.,  4.],
        [ 7.,  5.,  1.,  4.]])]
>>> hsplit(a,(3,4)) # Split a after the third and the fourth column
[array([[ 8.,  8.,  3.],
        [ 0.,  3.,  2.]]), array([[ 9.],
        [ 9.]])], array([[ 0.,  4.,  3.,  0.,  0.,  6.,  4.,  4.],
        [ 6.,  0.,  4.,  5.,  7.,  5.,  1.,  4.]])]
```

[vsplit](#) splits along the vertical axis, and [array_split](#) allows one to specify along which axis to split.

Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are three cases:

No Copy at All

Simple assignments make no copy of array objects or of their data.

```
>>> a = arange(12)
>>> b = a # no new object is created
>>> b is a # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4 # changes the shape of a
>>> a.shape
(3, 4)
```

Python passes mutable objects as references, so function calls make no copy.

```
>>> def f(x):
...     print id(x)
...
>>> id(a) # id is a unique identifier of an object
148293216
>>> f(a)
148293216
```

View or Shallow Copy

Different array objects can share the same data. The `view` method creates a new array object that looks at the same data.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6 # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0,4] = 1234 # a's data changes
>>> a
array([[ 0,  1,  2,  3],
       [1234,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Slicing an array returns a view of it:

```
>>> s = a[:, 1:3] # spaces added for clarity; could also be written "s = a[:,1:3]"
>>> s[:] = 10 # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[ 0, 10, 10,  3],
       [1234, 10, 10,  7],
       [ 8, 10, 10, 11]])
```


Deep Copy

The `copy` method makes a complete copy of the array and its data.

```
>>> d = a.copy() # a new array object with new data is created
>>> d is a
False
>>> d.base is a # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[ 0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

Functions and Methods Overview

Here is a list of NumPy functions and methods names ordered in some categories. The names link to the [Numpy_Example_List](#) so that you can see the functions in action.

Array Creation

[arange](#), [array](#), [copy](#), [empty](#), [empty_like](#), [eye](#), [fromfile](#), [fromfunction](#), [identity](#), [linspace](#), [logspace](#), [mgrid](#), [ogrid](#), [ones](#), [ones_like](#), [r](#), [zeros](#), [zeros_like](#)

Conversions

[astype](#), [atleast 1d](#), [atleast 2d](#), [atleast 3d](#), [mat](#)

Manipulations

[array split](#), [column stack](#), [concatenate](#), [diagonal](#), [dsplit](#), [dstack](#), [hsplit](#), [hstack](#), [item](#), [newaxis](#), [ravel](#), [repeat](#), [reshape](#), [resize](#), [squeeze](#), [swapaxes](#), [take](#), [transpose](#), [vsplit](#), [vstack](#)

Questions

all, any, nonzero, where

Ordering

[argmax](#), [argmin](#), [argsort](#), [max](#), [min](#), [ptp](#), [searchsorted](#), [sort](#)

Operations

[choose](#), [compress](#), [cumprod](#), [cumsum](#), [inner](#), [fill](#), [imag](#), [prod](#), [put](#), [putmask](#), [real](#), [sum](#)

Basic Statistics

cov, mean, std, var

Basic Linear Algebra

[cross](#), [dot](#), [outer](#), [svd](#), [vdot](#)

Less Basic

Broadcasting rules

Broadcasting allows universal functions to deal in a meaningful way with inputs that do not have exactly the same shape.

The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.

The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

After application of the broadcasting rules, the sizes of all arrays must match. More details can be found in [this documentation](#).

Fancy indexing and index tricks

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can be indexed by arrays of integers and arrays of booleans.

Indexing with Arrays of Indices

```
>>> a = arange(12)**2 # the first 12 square numbers
```

```
>>> i = array( [ 1,1,3,8,5 ] )           # an array of indices
>>> a[i]                                   # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = array( [ [ 3, 4], [ 9, 7 ] ] )     # a bidimensional array of indices
>>> a[j]                                   # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

When the indexed array `a` is multidimensional, a single array of indices refers to the first dimension of `a`. The following example shows this behavior by converting an image of labels into a color image using a palette.

```
>>> palette = array( [ [0,0,0],           # black
...                   [255,0,0],          # red
...                   [0,255,0],          # green
...                   [0,0,255],          # blue
...                   [255,255,255] ] )   # white
>>> image = array( [ [ 0, 1, 2, 0 ],      # each value corresponds to a color in the palette
...                [ 0, 3, 4, 0 ] ] )
>>> palette[image]                        # the (2,4,3) color image
array([[ [ 0,  0,  0],
        [255, 0,  0],
        [ 0, 255,  0],
        [ 0,  0,  0]],
       [ [ 0,  0,  0],
        [ 0,  0, 255],
        [255, 255, 255],
        [ 0,  0,  0]])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

```
>>> a = arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = array( [ [0,1],                   # indices for the first dim of a
...            [1,2] ] )
>>> j = array( [ [2,1],                   # indices for the second dim
...            [3,3] ] )
>>>
>>> a[i,j]                               # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]                               # i.e., a[ : , j]
array([[ [ 2,  1],
        [ 3,  3]],
       [ [ 6,  5],
        [ 7,  7]],
       [ [10,  9],
        [11, 11]])]
```

Naturally, we can put `i` and `j` in a sequence (say a list) and then do the indexing with the list.

```
>>> l = [i,j]
>>> a[l]                                  # equivalent to a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

However, we can not do this by putting `i` and `j` into an array, because this array will be interpreted as indexing the first dimension of `a`.

```
>>> s = array( [i,j] )
>>> a[s]                                  # not what we want
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index (3) out of range (0<=index<=2) in dimension 0
>>>
>>> a[tuple(s)]                           # same as a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

Another common use of indexing with arrays is the search of the maximum value of time-dependent series :

```
>>> time = linspace(20, 145, 5)           # time scale
>>> data = sin(arange(20)).reshape(5,4)    # 4 time-dependent series
>>> time
array([ 20.   ,  51.25,  82.5 , 113.75, 145.   ])
>>> data
array([[ 0.         ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])
>>>
>>> ind = data.argmax(axis=0)              # index of the maxima for each series
>>> ind
array([2, 0, 3, 1])
```

```
>>>
>>> time_max = time[ ind]                # times corresponding to the maxima
>>>
>>> data_max = data[ind, xrange(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...
>>>
>>> time_max
array([ 82.5 ,  20. , 113.75,  51.25])
>>> data_max
array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])
>>>
>>> all(data_max == data.max(axis=0))
True
```

You can also use indexing with arrays as a target to assign to:

```
>>> a = arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]] = 0
>>> a
array([0, 0, 2, 0, 0])
```

However, when the list of indices contains repetitions, the assignment is done several times, leaving behind the last value:

```
>>> a = arange(5)
>>> a[[0,0,2]]=1,2,3]
>>> a
array([2, 1, 3, 3, 4])
```

This is reasonable enough, but watch out if you want to use Python's += construct, as it may not do what you expect:

```
>>> a = arange(5)
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

Even though 0 occurs twice in the list of indices, the 0th element is only incremented once. This is because Python requires "a+=1" to be equivalent to "a=a+1".

Indexing with Boolean Arrays

When we index arrays with arrays of (integer) indices we are providing the list of indices to pick. With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.

The most natural way one can think of for boolean indexing is to use boolean arrays that have *the same shape* as the original array:

```
>>> a = arange(12).reshape(3,4)
>>> b = a > 4
>>> b
array([[False, False, False, False],
       [False, True, True, True],
       [True, True, True, True]], dtype=bool)
>>> a[b]
array([ 5,  6,  7,  8,  9, 10, 11])
```

This property can be very useful in assignments:

```
>>> a[b] = 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

You can look at the [Mandelbrot set example](#) to see how to use boolean indexing to generate an image of the [Mandelbrot set](#).

The second way of indexing with booleans is more similar to integer indexing; for each dimension of the array we give a 1D boolean array selecting the slices we want.

```
>>> a = arange(12).reshape(3,4)
>>> b1 = array([False,True,True])
>>> b2 = array([True,False,True,False])
>>>
>>> a[b1,:]
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]
array([ 4, 10])
```

Note that the length of the 1D boolean array must coincide with the length of the dimension (or axis) you want to slice. In the previous example, b1

is a 1-rank array with length 3 (the number of *rows* in *a*), and *b2* (of length 4) is suitable to index the 2nd rank (columns) of *a*.

The `ix_()` function

The `ix_` function can be used to combine different vectors so as to obtain the result for each n-uplet. For example, if you want to compute all the *a*+*b***c* for all the triplets taken from each of the vectors *a*, *b* and *c*:

```
>>> a = array([2,3,4,5])
>>> b = array([8,5,4])
>>> c = array([5,4,6,8,3])
>>> ax,bx,cx = ix_(a,b,c)
>>> ax
array([[2]],

       [[3]],

       [[4]],

       [[5]])
>>> bx
array([[8],

       [5],

       [4]])
>>> cx
array([[5, 4, 6, 8, 3]])
>>> ax.shape, bx.shape, cx.shape
((4, 1, 1), (1, 3, 1), (1, 1, 5))
>>> result = ax+bx*cx
>>> result
array([[42, 34, 50, 66, 26],
       [27, 22, 32, 42, 17],
       [22, 18, 26, 34, 14]],

       [[43, 35, 51, 67, 27],
       [28, 23, 33, 43, 18],
       [23, 19, 27, 35, 15]],

       [[44, 36, 52, 68, 28],
       [29, 24, 34, 44, 19],
       [24, 20, 28, 36, 16]],

       [[45, 37, 53, 69, 29],
       [30, 25, 35, 45, 20],
       [25, 21, 29, 37, 17]])
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

You could also implement the reduce as follows:

```
def ufunc_reduce(ufct, *vectors):
    vs = ix_(*vectors)
    r = ufct.identity
    for v in vs:
        r = ufct(r,v)
    return r
```

and then use it as:

```
>>> ufunc_reduce(add,a,b,c)
array([[15, 14, 16, 18, 13],
       [12, 11, 13, 15, 10],
       [11, 10, 12, 14, 9]],

       [[16, 15, 17, 19, 14],
       [13, 12, 14, 16, 11],
       [12, 11, 13, 15, 10]],

       [[17, 16, 18, 20, 15],
       [14, 13, 15, 17, 12],
       [13, 12, 14, 16, 11]],

       [[18, 17, 19, 21, 16],
       [15, 14, 16, 18, 13],
       [14, 13, 15, 17, 12]])
```

The advantage of this version of reduce compared to the normal `ufunc.reduce` is that it makes use of the [Broadcasting Rules](#) in order to avoid creating an argument array the size of the output times the number of vectors.

Indexing with strings

See [RecordArrays](#).

Linear Algebra

Work in progress. Basic linear algebra to be included here.

Simple Array Operations

See `linalg.py` in `numpy` folder for more.

```

>>> from numpy import *
>>> from numpy.linalg import *

>>> a = array([[1.0, 2.0], [3.0, 4.0]])
>>> print a
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = array([[0.0, -1.0], [1.0, 0.0]])

>>> dot (j, j) # matrix product
array([[-1.,  0.],
       [ 0., -1.]])

>>> trace(u) # trace
2.0

>>> y = array([[5.], [7.]])
>>> solve(a, y)
array([[-3.],
       [ 4.]])

>>> eig(j)
(array([ 0.+1.j,  0.-1.j]),
array([[ 0.70710678+0.j,  0.70710678+0.j],
       [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]]))
Parameters:
  square matrix

Returns
  The eigenvalues, each repeated according to its multiplicity.

  The normalized (unit "length") eigenvectors, such that the
  column ``v[:,i]`` is the eigenvector corresponding to the
  eigenvalue ``w[i]``.

```

The Matrix Class

Here is a short intro to the Matrix class.

```

>>> A = matrix('1.0 2.0; 3.0 4.0')
>>> A
[[ 1.  2.]
 [ 3.  4.]]
>>> type(A) # file where class is defined
<class 'numpy.matrixlib.defmatrix.matrix'>

>>> A.T # transpose
[[ 1.  3.]
 [ 2.  4.]]

>>> X = matrix('5.0 7.0')
>>> Y = X.T
>>> Y
[[5.]
 [7.]]

>>> print A*Y # matrix multiplication
[[19.]
 [43.]]

>>> print A.I # inverse
[[-2.  1.]
 [ 1.5 -0.5]]

>>> solve(A, Y) # solving linear equation
matrix([[-3.],
        [ 4.]])

```

Indexing: Comparing Matrices and 2D Arrays

Note that there are some important differences between [NumPy](#) arrays and matrices. [NumPy](#) provides two fundamental objects: an N-dimensional array object and a universal function object. Other objects are built on top of these. In particular, matrices are 2-dimensional array objects that inherit from the [NumPy](#) array object. For both arrays and matrices, indices must consist of a proper combination of one or more of the following: integer scalars, ellipses, a list of integers or boolean values, a tuple of integers or boolean values, and a 1-dimensional array of integer or boolean values. A matrix can be used as an index for matrices, but commonly an array, list, or other form is needed to accomplish a given task.

As usual in Python, indexing is zero-based. Traditionally we represent a 2D array or matrix as a rectangular array of rows and columns, where

movement along axis 0 is movement across rows, while movement along axis 1 is movement across columns.

Let's make an array and matrix to slice:

```
>>> A = arange(12)
>>> A
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> A.shape = (3,4)
>>> M = mat(A.copy())
>>> print type(A), " ", type(M)
<type 'numpy.ndarray'>      <class 'numpy.core.defmatrix.matrix'>
>>> print A
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print M
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Now, let's take some simple slices. Basic slicing uses slice objects or integers. For example, the evaluation of `A[:]` and `M[:]` will appear familiar from Python indexing, however it is important to note that slicing [NumPy](#) arrays does **not** make a copy of the data; slicing provides a new view of the same data.

```
>>> print A[:]; print A[:].shape
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
(3, 4)
>>> print M[:]; print M[:].shape
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
(3, 4)
```

Now for something that differs from Python indexing: you may use comma-separated indices to index along multiple axes at the same time.

```
>>> print A[:,1]; print A[:,1].shape
[1 5 9]
(3,)
>>> print M[:,1]; print M[:,1].shape
[[1]
 [5]
 [9]]
(3, 1)
```

Notice the difference in the last two results. Use of a single colon for the 2D array produces a 1-dimensional array, while for a matrix it produces a 2-dimensional matrix. A slice of a matrix will always produce a matrix. For example, a slice `M[2,:]` produces a matrix of shape (1,4). In contrast, a slice of an array will always produce an array of the lowest possible dimension. For example, if `C` were a 3-dimensional array, `C[... ,1]` produces a 2D array while `C[1,: ,1]` produces a 1-dimensional array. From this point on, we will show results only for the array slice if the results for the corresponding matrix slice are identical.

Lets say that we wanted the 1st and 3rd column of an array. One way is to slice using a list:

```
>>> A[:, [1,3]]
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

A slightly more complicated way is to use the `take()` method:

```
>>> A[:,].take([1,3],axis=1)
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

If we wanted to skip the first row, we could use:

```
>>> A[1:,:].take([1,3],axis=1)
array([[ 5,  7],
       [ 9, 11]])
```

Or we could simply use `A[1:,[1,3]]`. Yet another way to slice the above is to use a cross product:

```
>>> A[ix_((1,2),(1,3)) ]
array([[ 5,  7],
       [ 9, 11]])
```

For the reader's convenience, here is our array again:

```
>>> print A
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Now let's do something a bit more complicated. Lets say that we want to retain all columns where the first row is greater than 1. One way is to create a boolean index:

```
>>> A[0,:]>1
array([False, False, True, True], dtype=bool)
>>> A[:,A[0,:]>1]
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

Just what we wanted! But indexing the matrix is not so convenient.

```
>>> M[0,:]>1
matrix([[False, False, True, True]], dtype=bool)
>>> M[:,M[0,:]>1]
matrix([[2, 3]])
```

The problem of course is that slicing the matrix slice produced a matrix. But matrices have a convenient 'A' attribute whose value is the array representation, so we can just do this instead:

```
>>> M[:,M.A[0,:]>1]
matrix([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

If we wanted to conditionally slice the matrix in two directions, we must adjust our strategy slightly. Instead of

```
>>> A[A[:,0]>2,A[0,:]>1]
array([ 6, 11])
>>> M[M.A[:,0]>2,M.A[0,:]>1]
matrix([[ 6, 11]])
```

we need to use the cross product 'ix_':

```
>>> A[numpy.ix_(A[:,0]>2,A[0,:]>1)]
array([[ 6,  7],
       [10, 11]])
>>> M[numpy.ix_(M.A[:,0]>2,M.A[0,:]>1)]
matrix([[ 6,  7],
       [10, 11]])
```

Tricks and Tips

Here we give a list of short and useful tips.

"Automatic" Reshaping

To change the dimensions of an array, you can omit one of the sizes which will then be deduced automatically:

```
>>> a = arange(30)
>>> a.shape = 2,-1,3 # -1 means "whatever is needed"
>>> a.shape
(2, 5, 3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23],
       [24, 25, 26],
       [27, 28, 29]])
```

Vector Stacking

How do we construct a 2D array from a list of equally-sized row vectors? In MATLAB this is quite easy: if `x` and `y` are two vectors of the same length you only need do `m=[x;y]`. In NumPy this works via the functions `column_stack`, `dstack`, `hstack` and `vstack`, depending on the dimension in which the stacking is to be done. For example:

```
x = arange(0,10,2) # x=([0,2,4,6,8])
y = arange(5)      # y=([0,1,2,3,4])
m = vstack([x,y])  # m=([ [0,2,4,6,8],
                    #      [0,1,2,3,4]])
xy = hstack([x,y]) # xy = ([0,2,4,6,8,0,1,2,3,4])
```

The logic behind those functions in more than two dimensions can be strange.

See also [NumPy for Matlab Users](#) and add your new findings there. ☐

Histograms


The NumPy `histogram` function applied to an array returns a pair of vectors: the histogram of the array and the vector of bins. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist` plots the histogram automatically, while `numpy.histogram` only generates the data.

```
import numpy
import pylab
# Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
mu, sigma = 2, 0.5
v = numpy.random.normal(mu,sigma,10000)
# Plot a normalized histogram with 50 bins
pylab.hist(v, bins=50, normed=1)          # matplotlib version (plot)
pylab.show()
# Compute the histogram with numpy and then plot it
(n, bins) = numpy.histogram(v, bins=50, normed=True) # NumPy version (no plot)
pylab.plot(.5*(bins[1:]+bins[:-1]), n)
pylab.show()
```

References

- The [Python tutorial](#).
- The [Numpy Example List](#).
- The nonexistent [NumPy Tutorial](#) at scipy.org, where we can find the old Numeric documentation.
- The [Guide to NumPy](#) book.
- The [SciPy Tutorial](#) and a [SciPy course online](#)
- [NumPy for Matlab Users](#).
- A [matlab, R, IDL, NumPy/SciPy dictionary](#).

SciPy: Tentative_NumPy_Tutorial (last edited 2015-10-24 17:48:27 by anonymous)

- [Attachments](#)
- More Actions: 
- [MoinMoin Powered](#)
- [Python Powered](#)
- [GPL licensed](#)
- [Valid HTML 4.01](#)