AI 고급프로그래밍

Numpy/matplotlib, pdb
Pandas
Data preprocessing, Template matching, k-NN, PCA & LDA
Perceptron and SVM
Neural Network(Multi-Layer Perceptron)
Convolution Neural Network
Tensorflow

Term-Project

프로그램 설치

Python: https://www.pythn.org Python 3.8.7

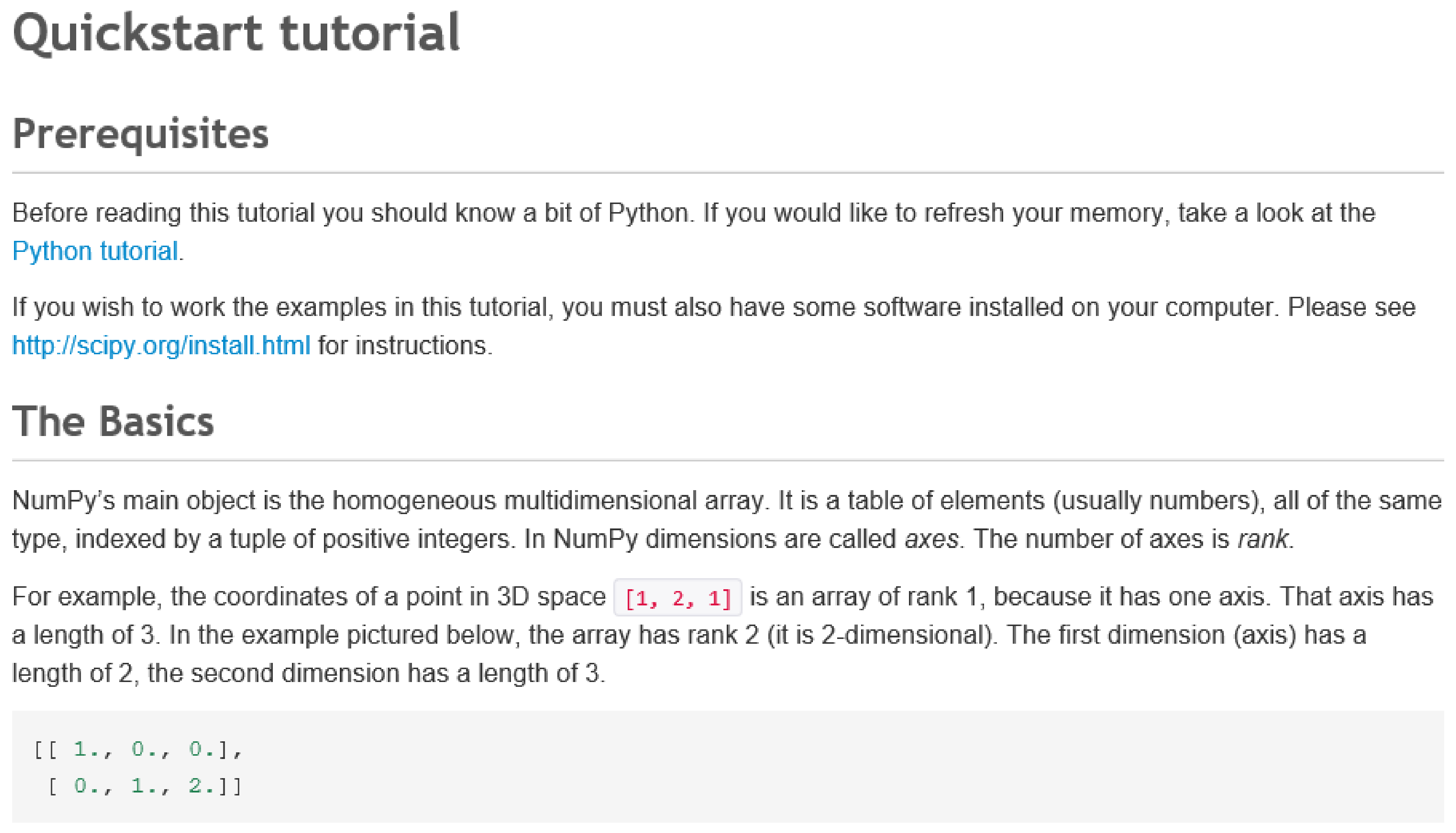
```
C:₩Users₩USER₩AppData₩Local₩Programs₩Python₩Python38>
```

Numpy설치 python –m pip install numpy
Matplotlib설치 python –m pip install matplotlib
Pandas설치 python –m pip install pandas
Pillow 설치 python –m pip install pillow (pil에서 pillow로 바뀜)
Scikit-learn 설치 python –m pip install scikit-learn
Tesnorflow 설치 python -m pip install tensorflow-cpu
(numpy, scipy, keras 가 같이 설치 됨)

프로그램 설치 후 import ... (예 import numpy)

Numpy

https://docs.scipy.org/doc/numpy-1.13.0/user/quickstart.html



Quickstart tutorial

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. Please see http://scipy.org/install.html for instructions.

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 the 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

```
>>> import numpy as np
>>> a = np.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
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

Array Creation

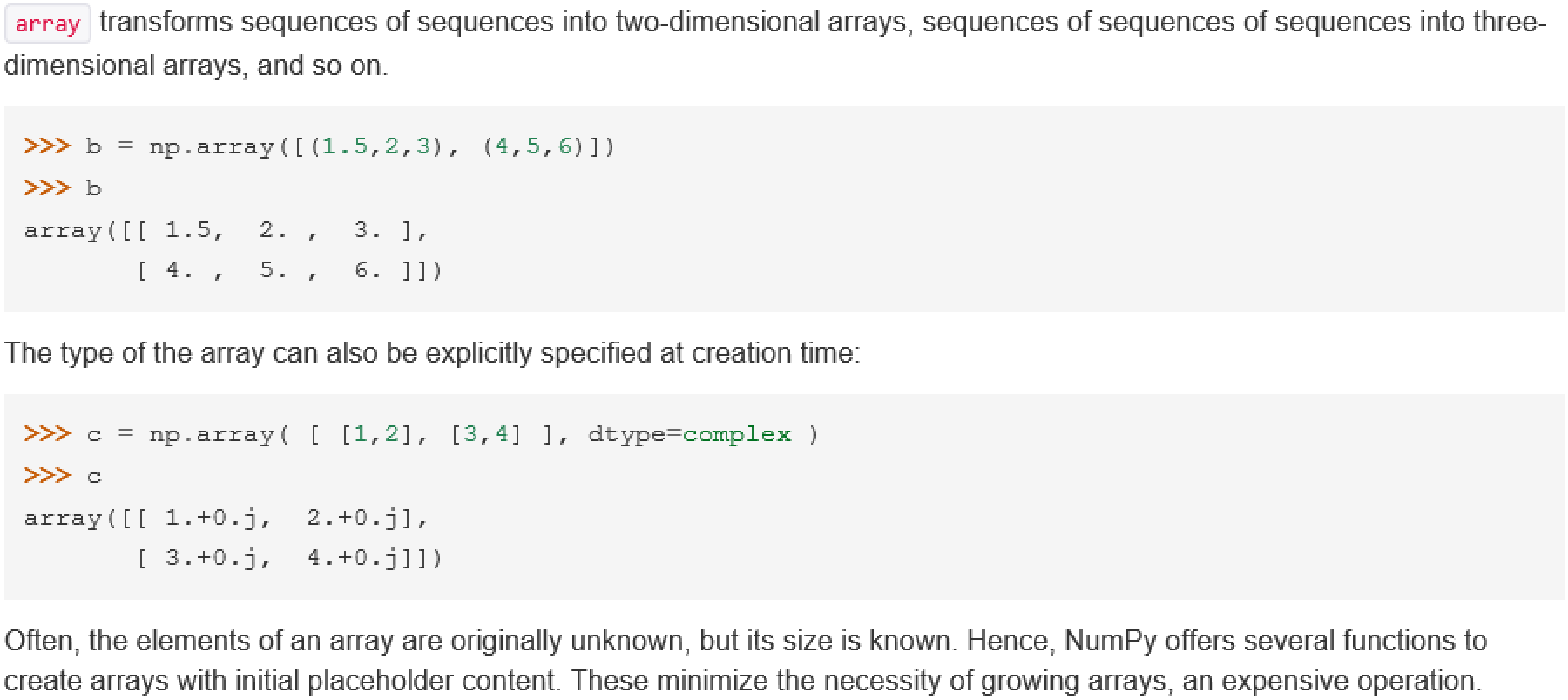
There are several ways to create arrays.

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.

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.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 = np.array(1,2,3,4)    # WRONG
>>> a = np.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 = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

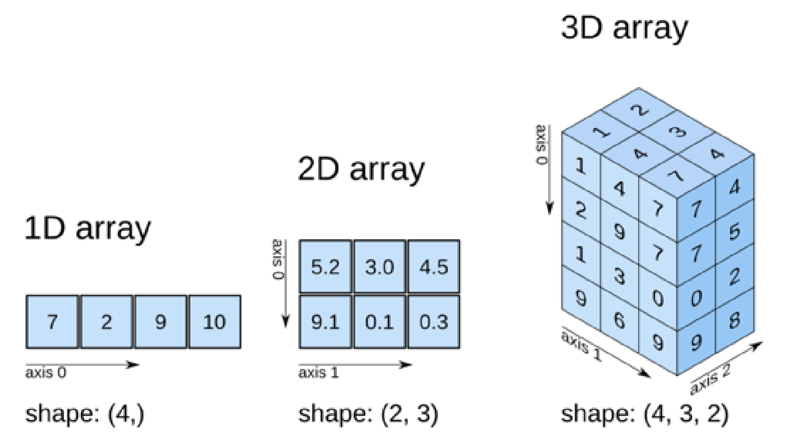
The type of the array can also be explicitly specified at creation time:

```
>>> c = np.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`.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )
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)
>>> np.empty( (2,3) )                                    # uninitialized, output may vary
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.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.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:

```
>>> from numpy import pi
>>> np.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 = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of points
>>> f = np.sin(x)
```

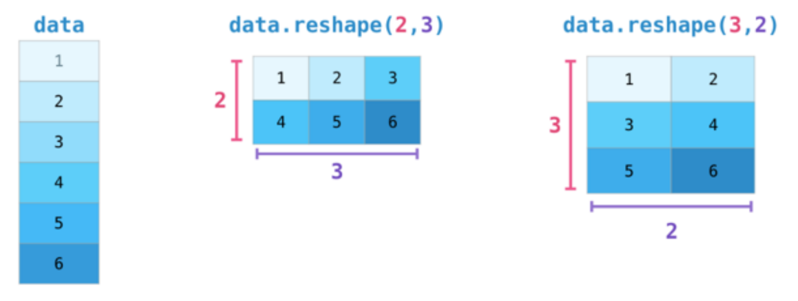
Printing Arrays¶

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 = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.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(np.arange(10000))
[   0    1    2 ..., 9997 9998 9999]
>>>
>>> print(np.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`.

```
>>> np.set_printoptions(threshold=np.nan)
```

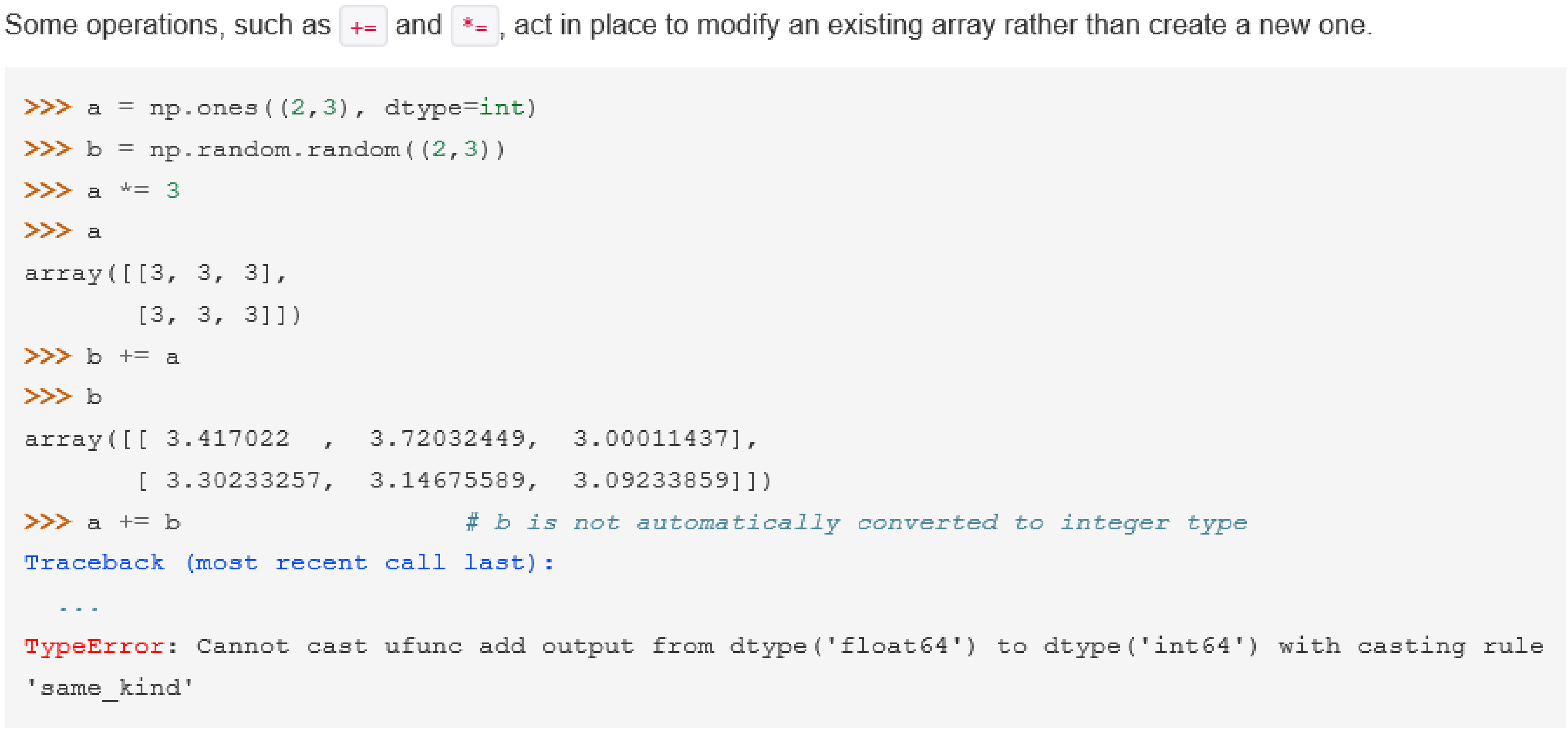
Basic Operations

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

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.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*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True, True, False, False], dtype=bool)
```

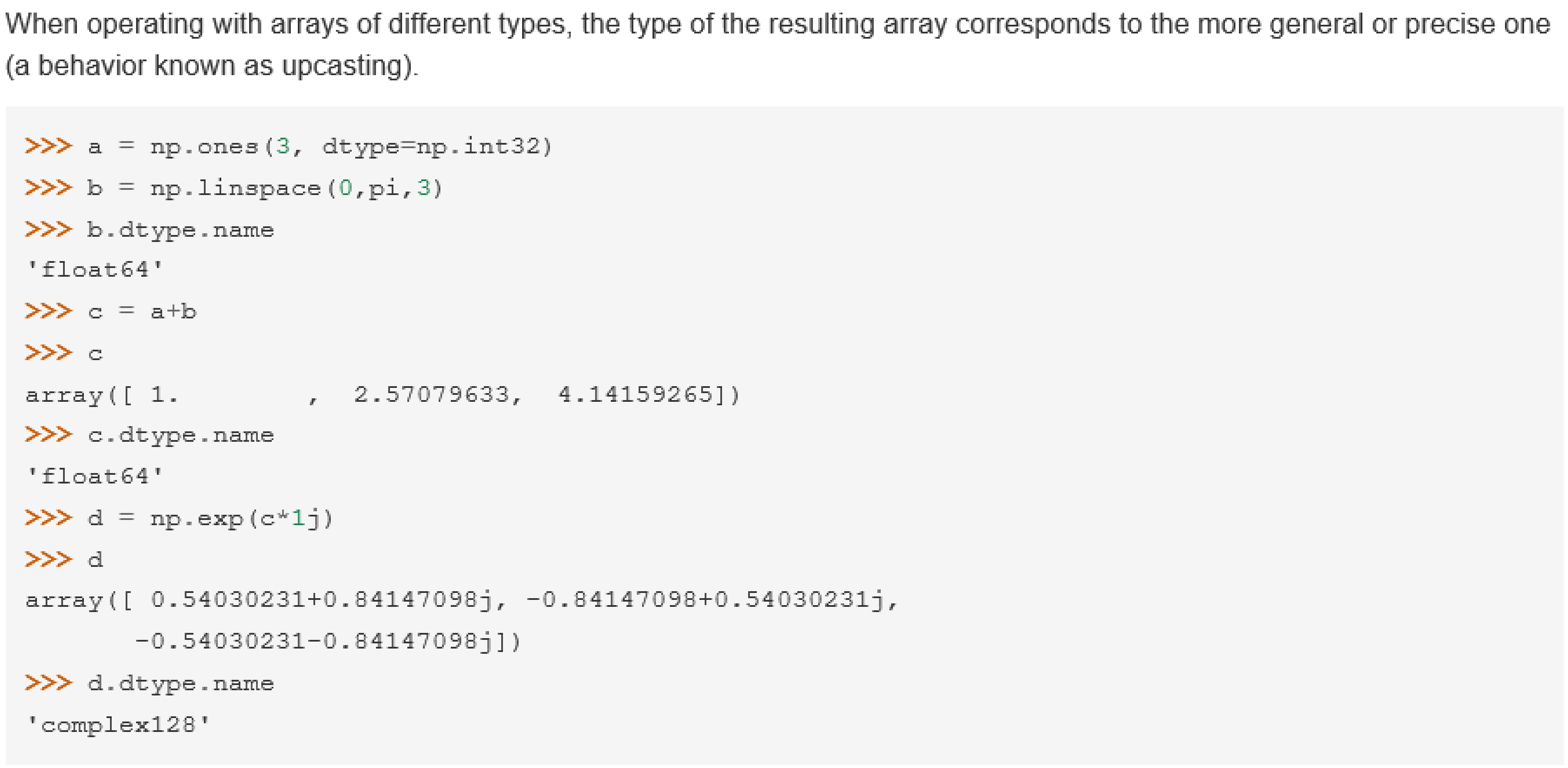
Unlike in many matrix languages, the product operator `*` operates elementwise in NumPy arrays. The matrix product can be performed using the `dot` function or method:

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



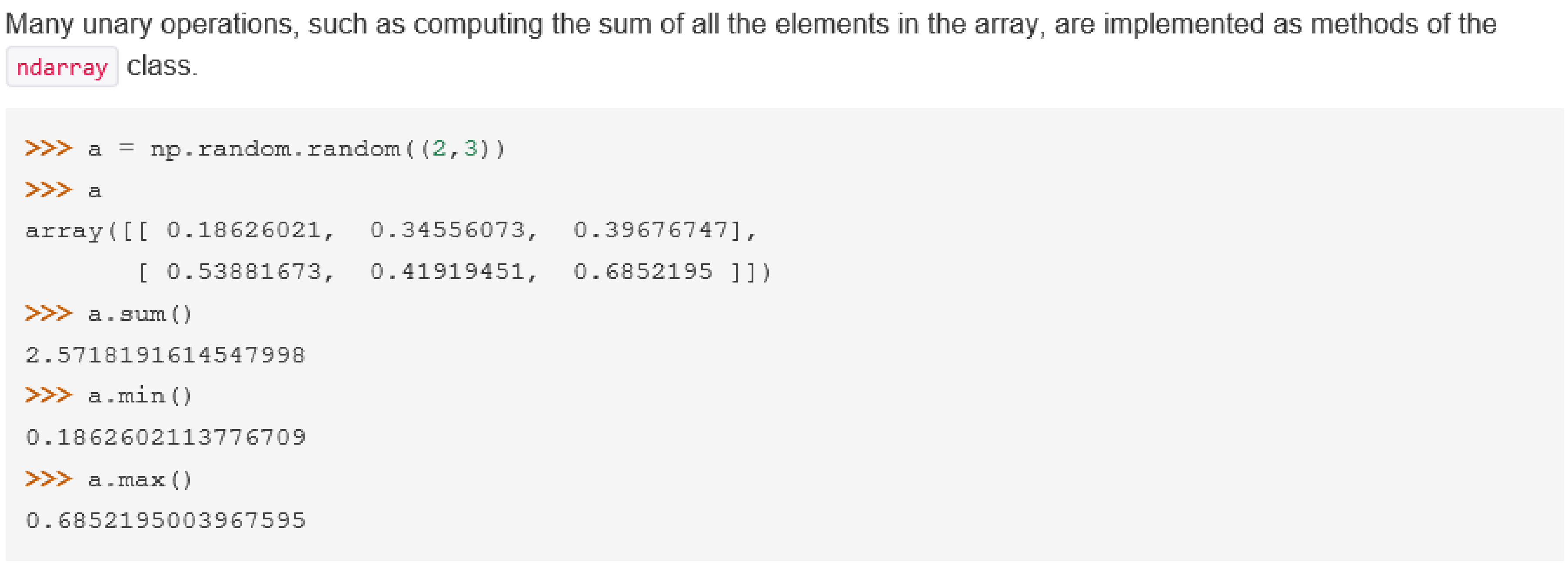
Some operations, such as `+=` and `*=`, act in place to modify an existing array rather than create a new one.

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
>>> a += b                  # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with casting rule
'same_kind'
```



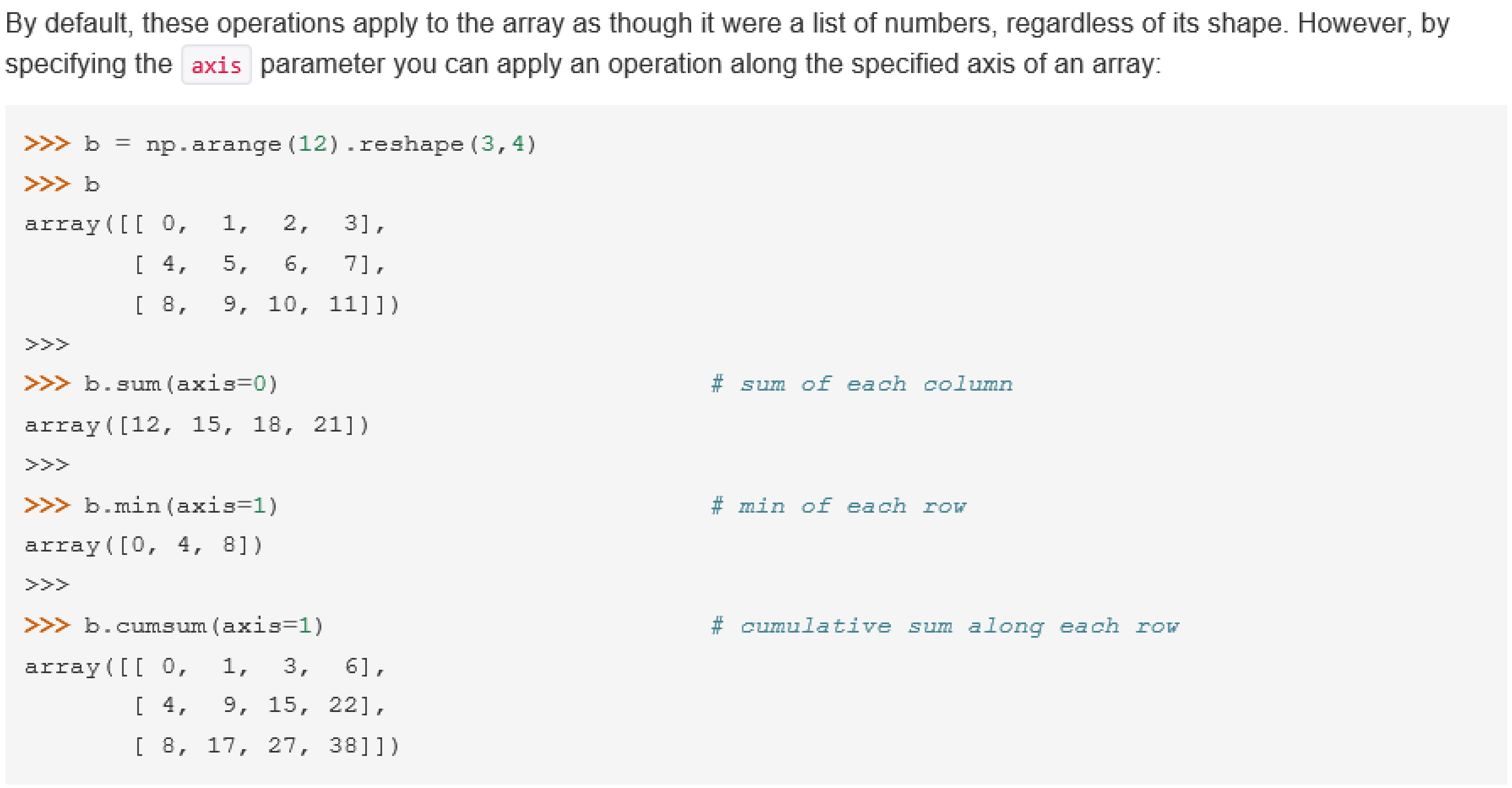
When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```



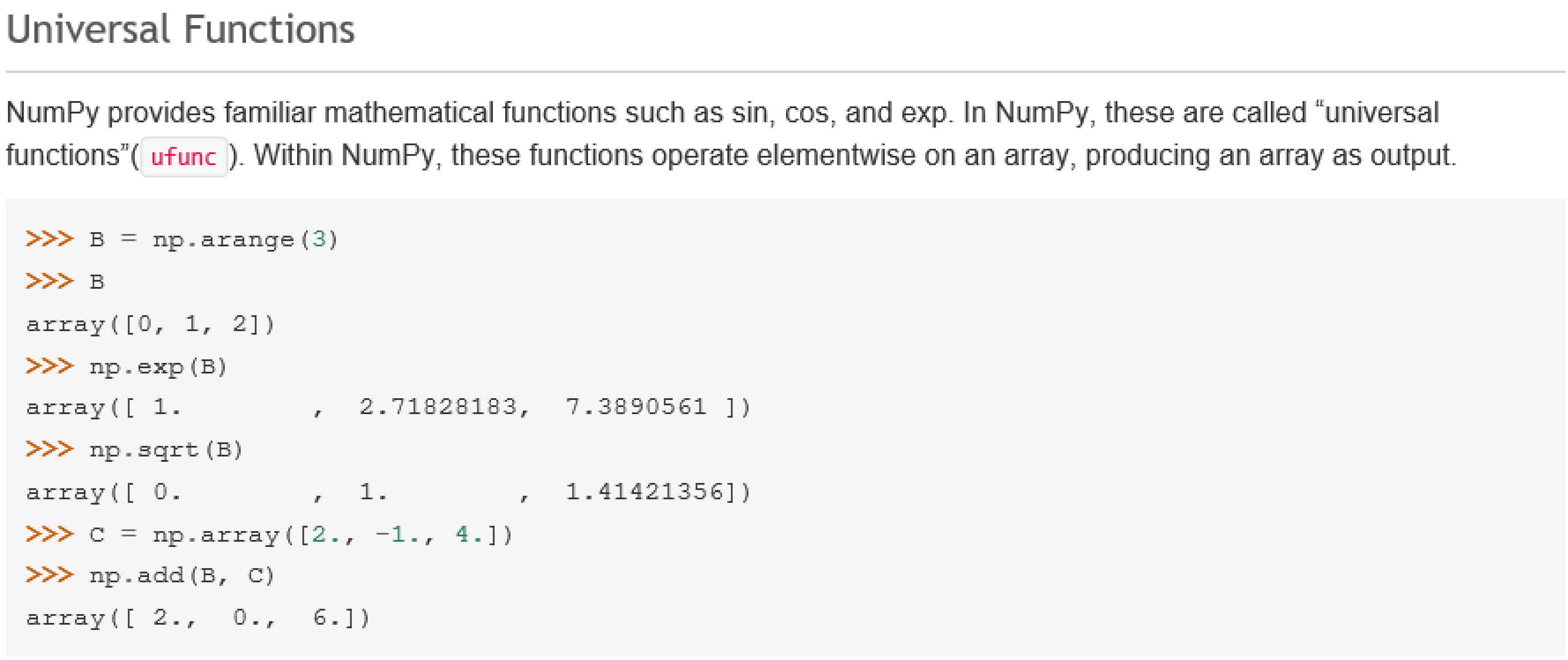
Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the `ndarray` class.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```



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:

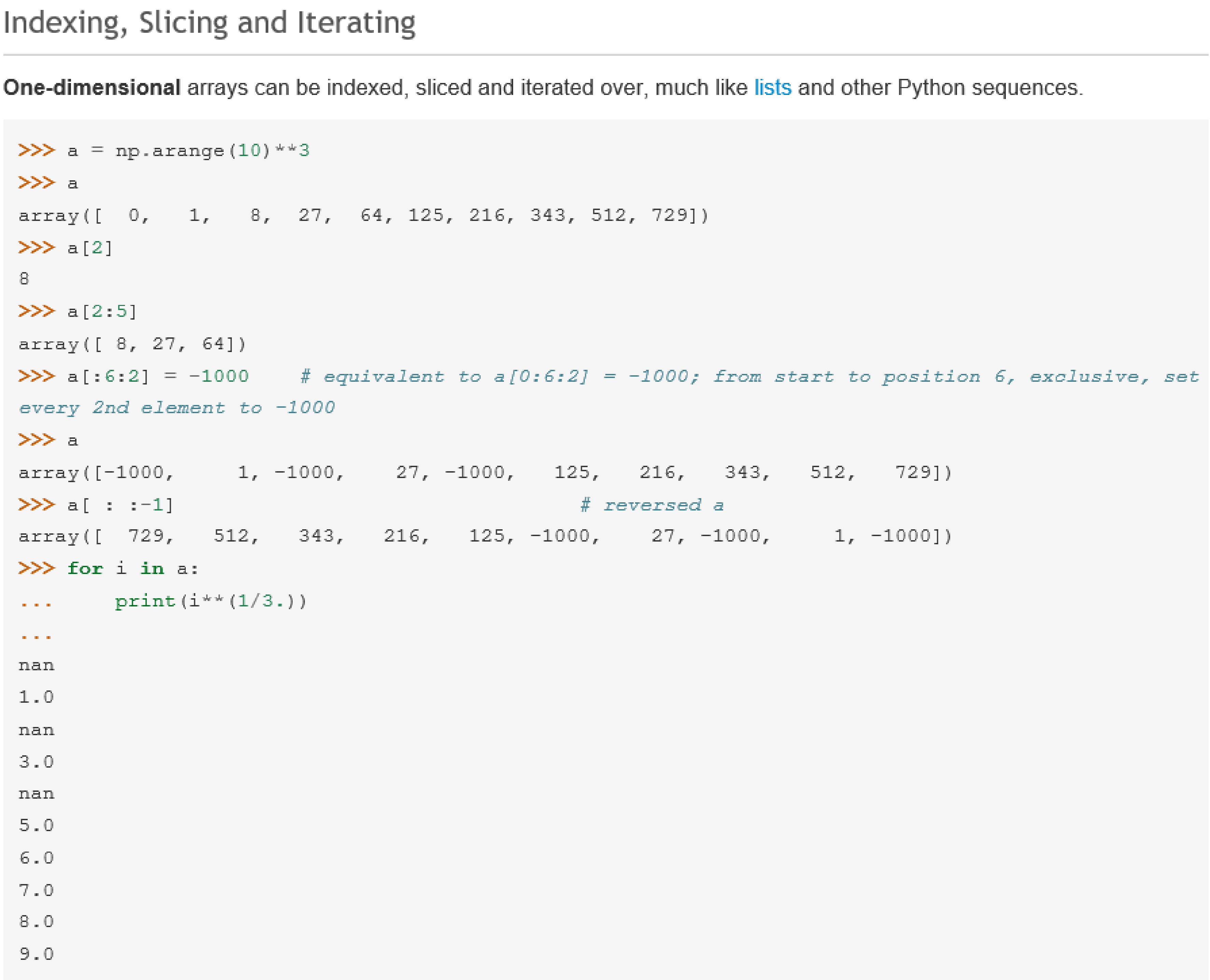
```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>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

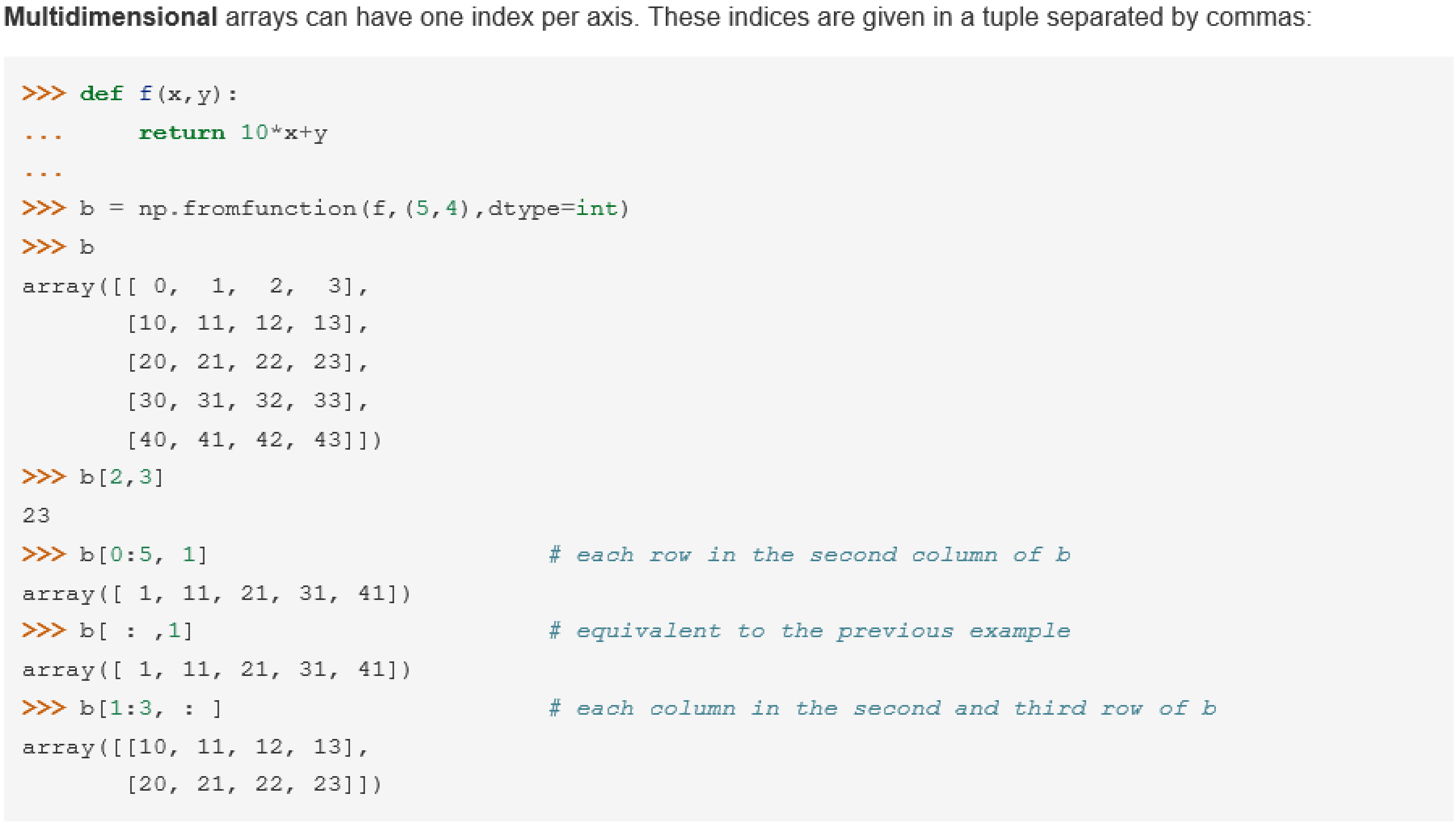
```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```



Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> a = np.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 = np.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,:]`.

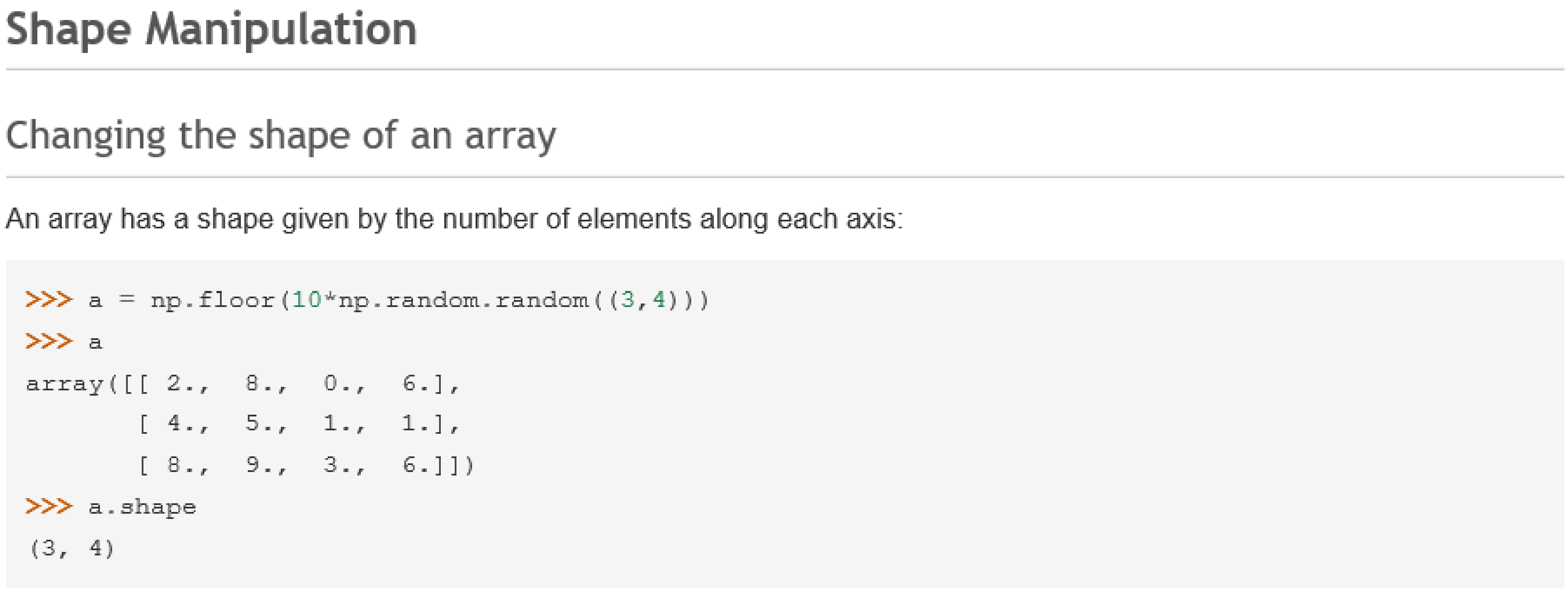
```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D arrays)
...                  [ 10, 12, 13]],
...                 [[100,101,102],
...                  [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100,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



Shape Manipulation

Changing the shape of an array

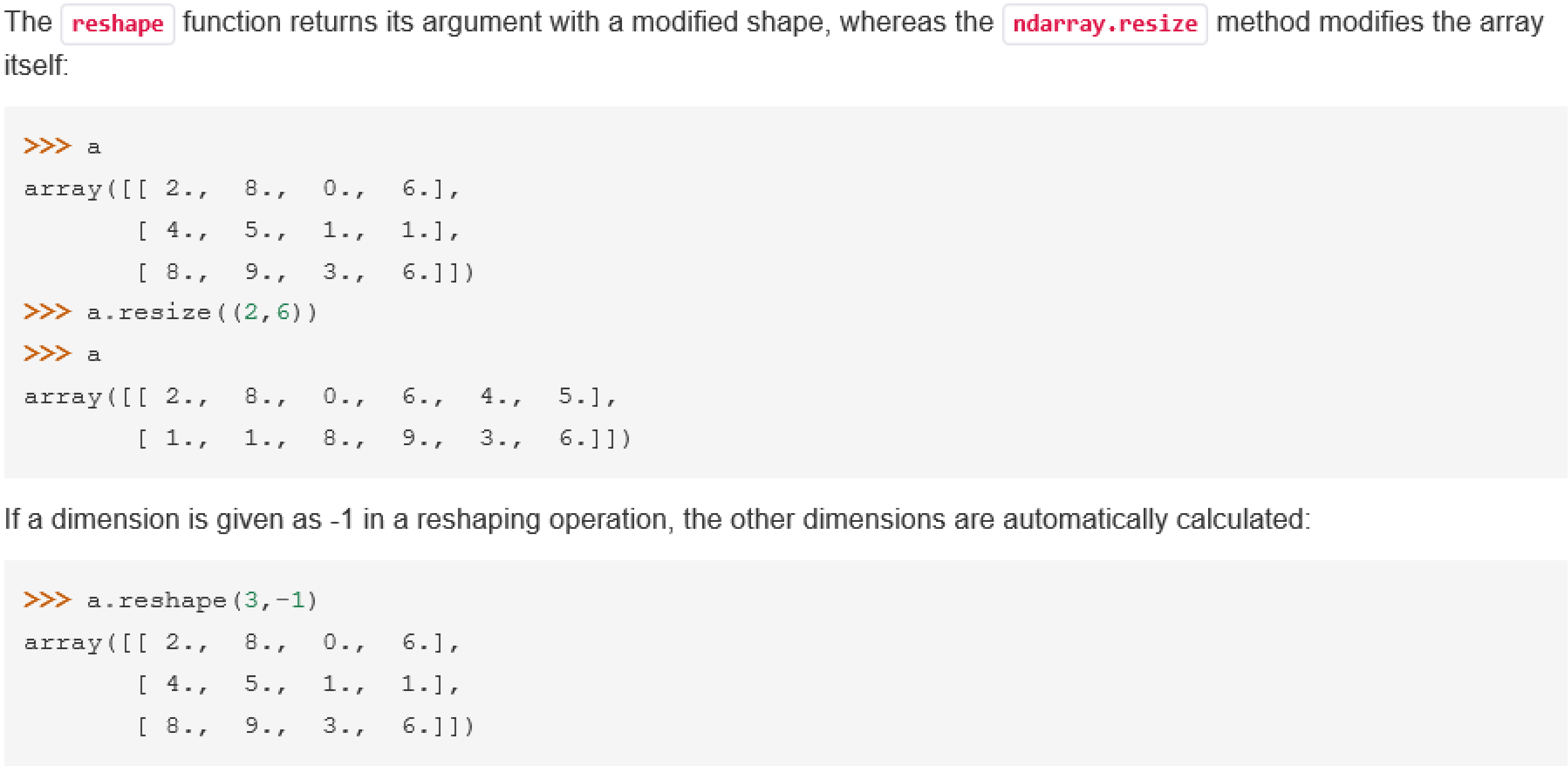
An array has a shape given by the number of elements along each axis:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

```
>>> a.ravel()  # returns the array, flattened
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

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.

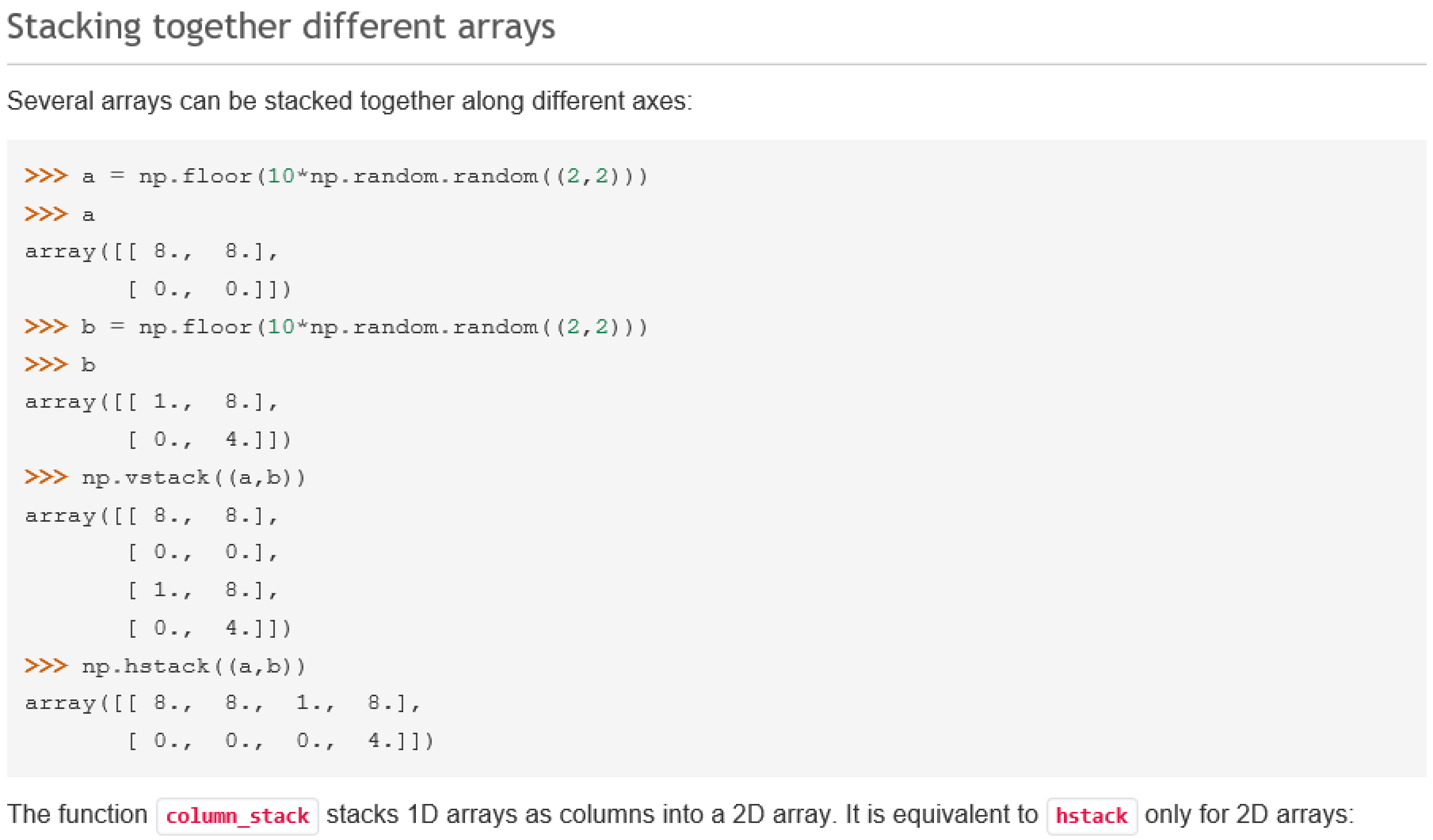


The `reshape` function returns its argument with a modified shape, whereas the `ndarray.resize` method modifies the array itself:

```
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

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

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```



Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

The function `column_stack` stacks 1D arrays as columns into a 2D array. It is equivalent to `hstack` only for 2D arrays:

The function `column_stack` stacks 1D arrays as columns into a 2D array. It is equivalent to `hstack` only for 2D arrays:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[ 4., 3.],
       [ 2., 8.]])
>>> np.hstack((a,b))           # the result is different
array([ 4., 2., 3., 8.])
>>> a[:,newaxis]               # this allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  3.],
       [ 2.,  8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[ 4.,  3.],
       [ 2.,  8.]])
```

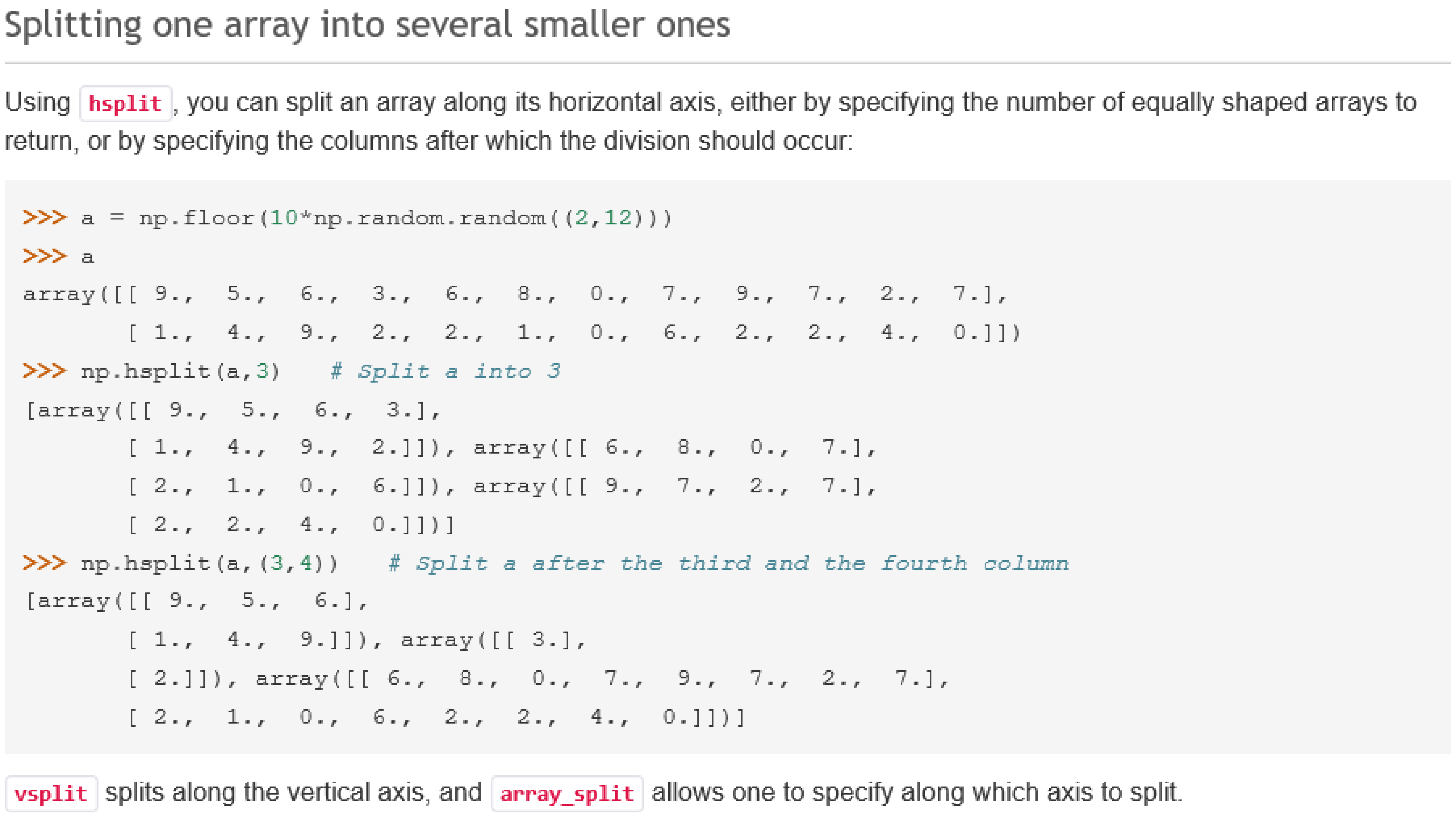
On the other hand, the function `row_stack` is equivalent to `vstack` for any input arrays. In general, for arrays of with more than two dimensions, `hstack` stacks along their second axes, `vstack` stacks along their first axes, and `concatenate` allows for an optional arguments giving the number of 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 (”:”)

```
>>> np.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.



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

`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:

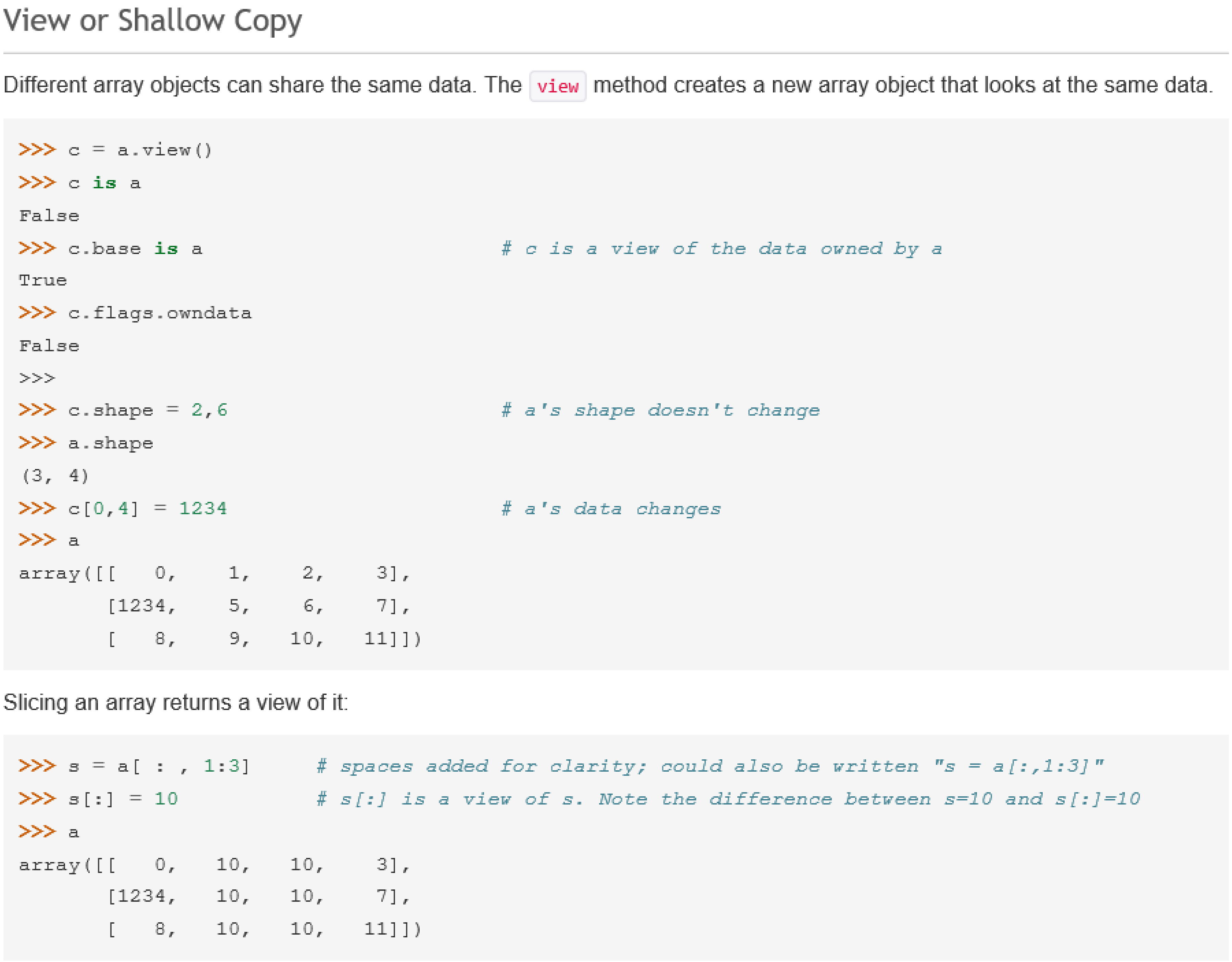
No Copy at All

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

```
>>> a = np.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
```



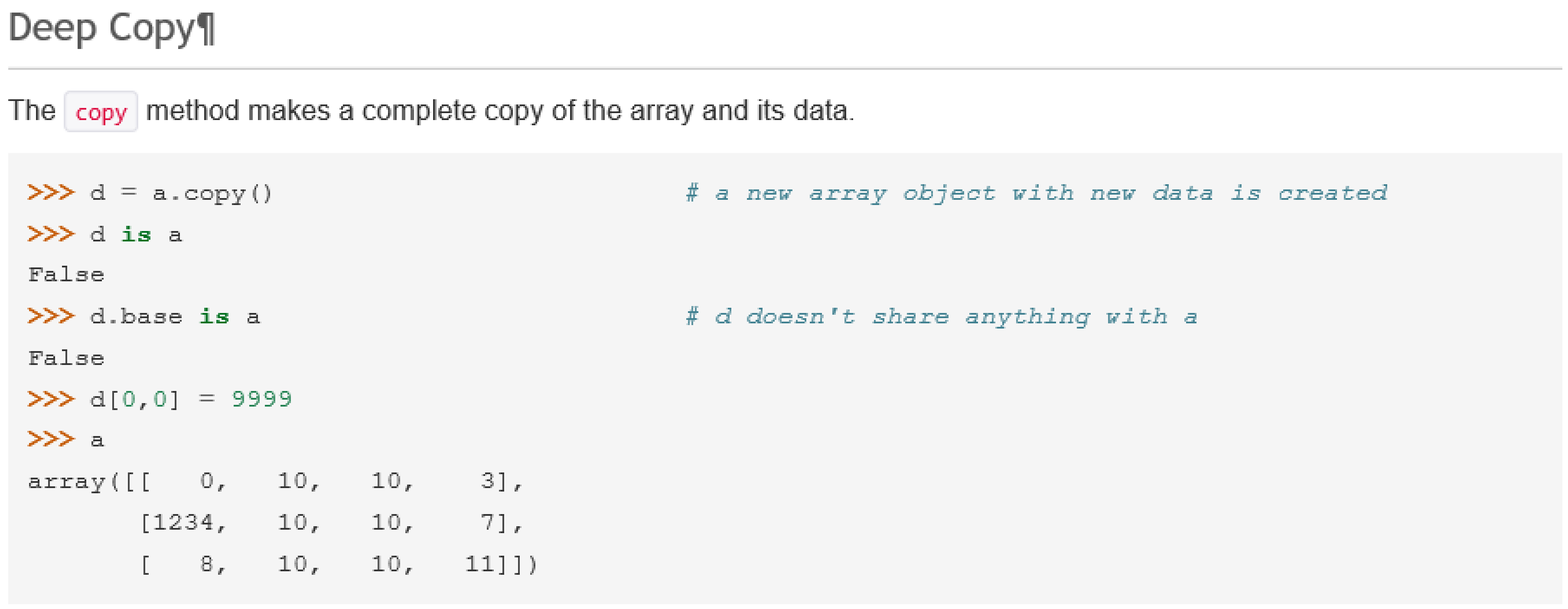
View or Shallow Copy

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:

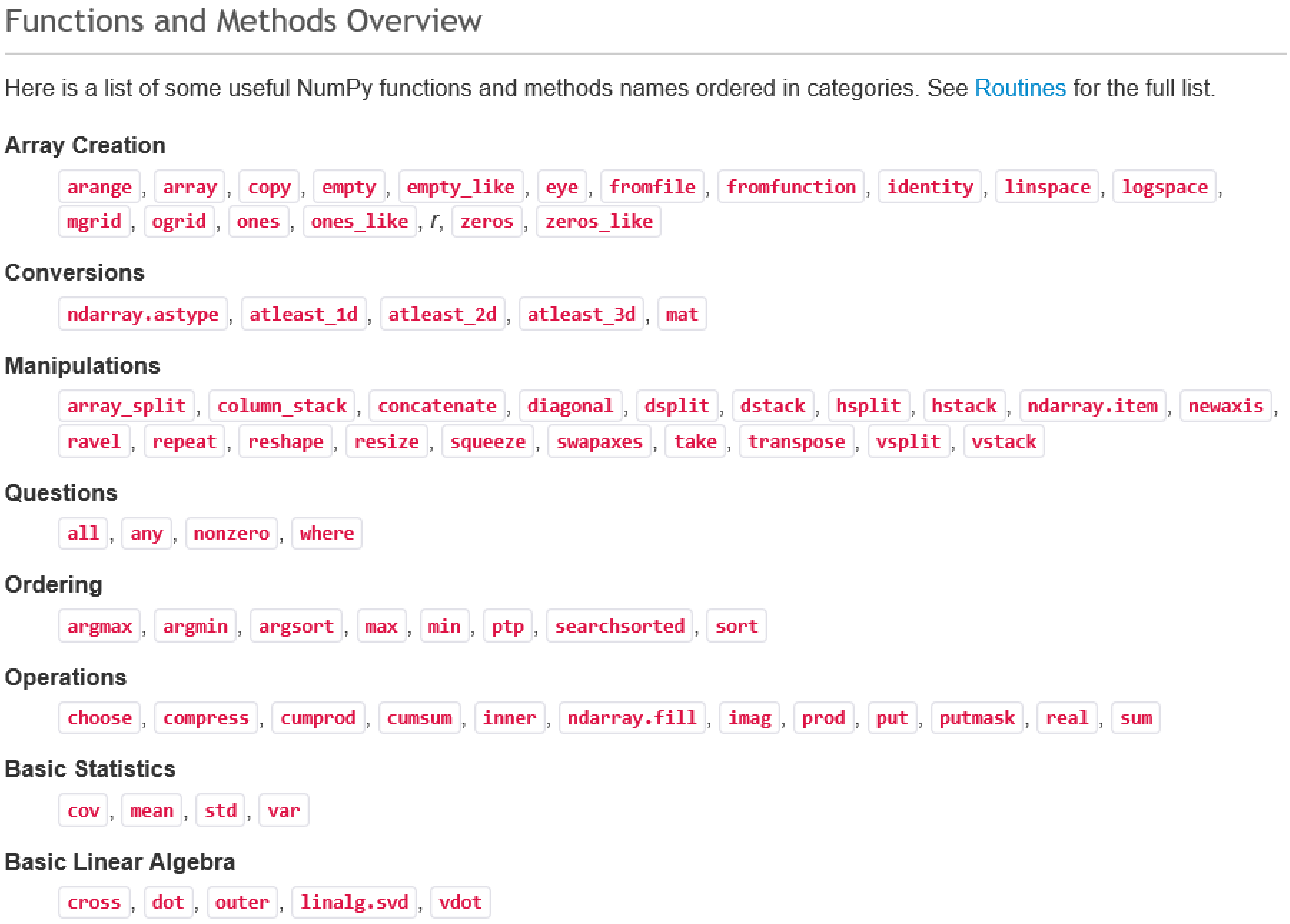
```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```



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 some useful NumPy functions and methods names ordered in categories. See Routines for the full list.

Array Creation

`arange`, `array`, `copy`, `empty`, `empty_like`, `eye`, `fromfile`, `fromfunction`, `identity`, `linspace`, `logspace`, `mgrid`, `ogrid`, `ones`, `ones_like`, *r*, `zeros`, `zeros_like`

Conversions

`ndarray.astype`, `atleast_1d`, `atleast_2d`, `atleast_3d`, `mat`

Manipulations

`array_split`, `column_stack`, `concatenate`, `diagonal`, `dsplit`, `dstack`, `hsplit`, `hstack`, `ndarray.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

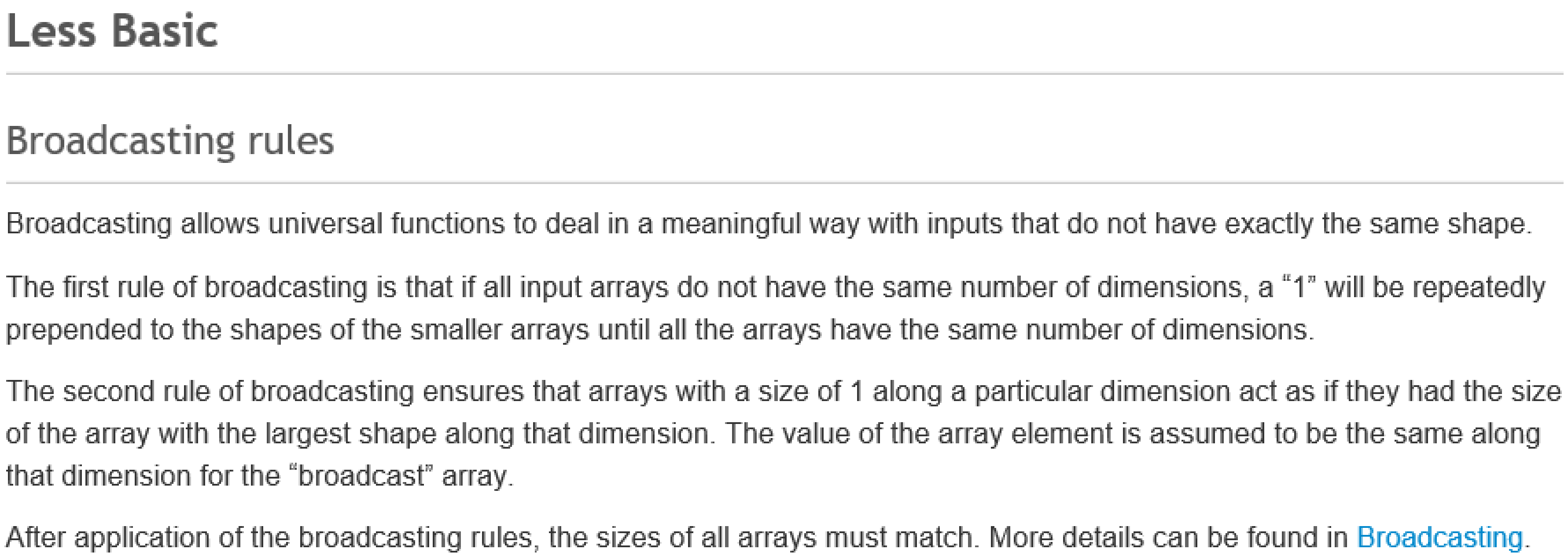
`choose`, `compress`, `cumprod`, `cumsum`, `inner`, `ndarray.fill`, `imag`, `prod`, `put`, `putmask`, `real`, `sum`

Basic Statistics

`cov`, `mean`, `std`, `var`

Basic Linear Algebra

`cross`, `dot`, `outer`, `linalg.svd`, `vdot`



Less Basic

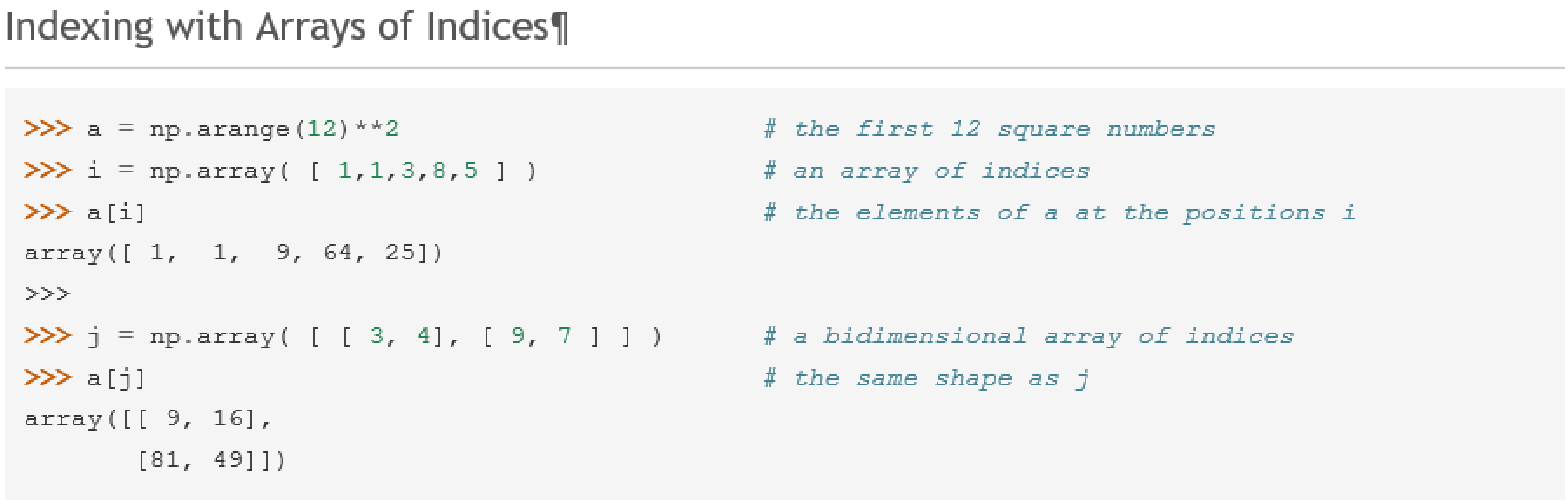
Broadcasting rules

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 Broadcasting.



Indexing with Arrays of Indices¶

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.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 = np.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 = np.array( [ [0,0,0],                # black
...                       [255,0,0],              # red
...                       [0,255,0],              # green
...                       [0,0,255],              # blue
...                       [255,255,255] ] )       # white
>>> image = np.array( [ [ 0, 1, 2, 0 ],           # each value corresponds to a color in the pale
tte
...                     [ 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 = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array( [ [0,1],                        # indices for the first dim of a
...                 [1,2] ] )
>>> j = np.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 = np.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 = np.linspace(20, 145, 5)                 # time scale
>>> data = np.sin(np.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 ])
>>>
>>> np.all(data_max == data.max(axis=0))
True
```

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

```
>>> a = np.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 = np.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 = np.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 = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                          # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> a[b]                                       # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

This property can be very useful in assignments:

```
>>> a[b] = 0                                   # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

You can look at the following example to see how to use boolean indexing to generate an image of the Mandelbrot set:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def mandelbrot( h,w, maxit=20 ):
...     """Returns an image of the Mandelbrot fractal of size (h,w)."""
...     y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
...     c = x+y*1j
...     z = c
...     divtime = maxit + np.zeros(z.shape, dtype=int)
...
...     for i in range(maxit):
...         z = z**2 + c
...         diverge = z*np.conj(z) > 2**2            # who is diverging
...         div_now = diverge & (divtime==maxit)  # who is diverging now
...         divtime[div_now] = i                  # note when
...         z[diverge] = 2                        # avoid diverging too much
...
...     return divtime
>>> plt.imshow(mandelbrot(400,400))
>>> plt.show()
```

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 = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True])             # first dim selection
>>> b2 = np.array([True,False,True,False])       # second dim selection
>>>
>>> a[b1,:]                                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                     # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]                                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                  # a weird thing to do
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 = np.array([2,3,4,5])
>>> b = np.array([8,5,4])
>>> c = np.array([5,4,6,8,3])
>>> ax,bx,cx = np.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
```

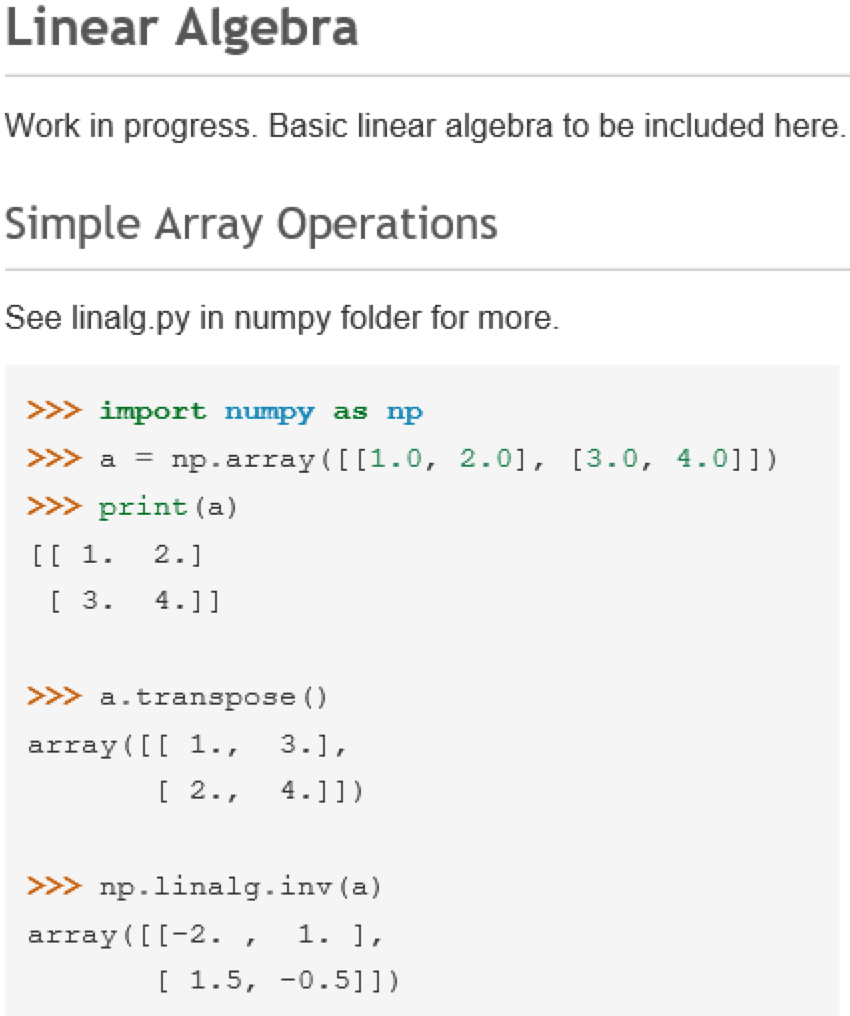
You could also implement the reduce as follows:

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

and then use it as:

```
>>> ufunc_reduce(np.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.



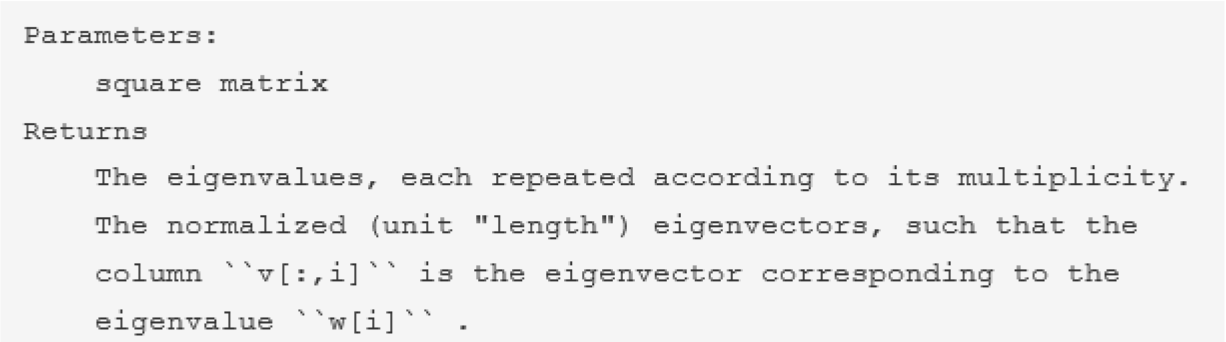
Linear Algebra

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

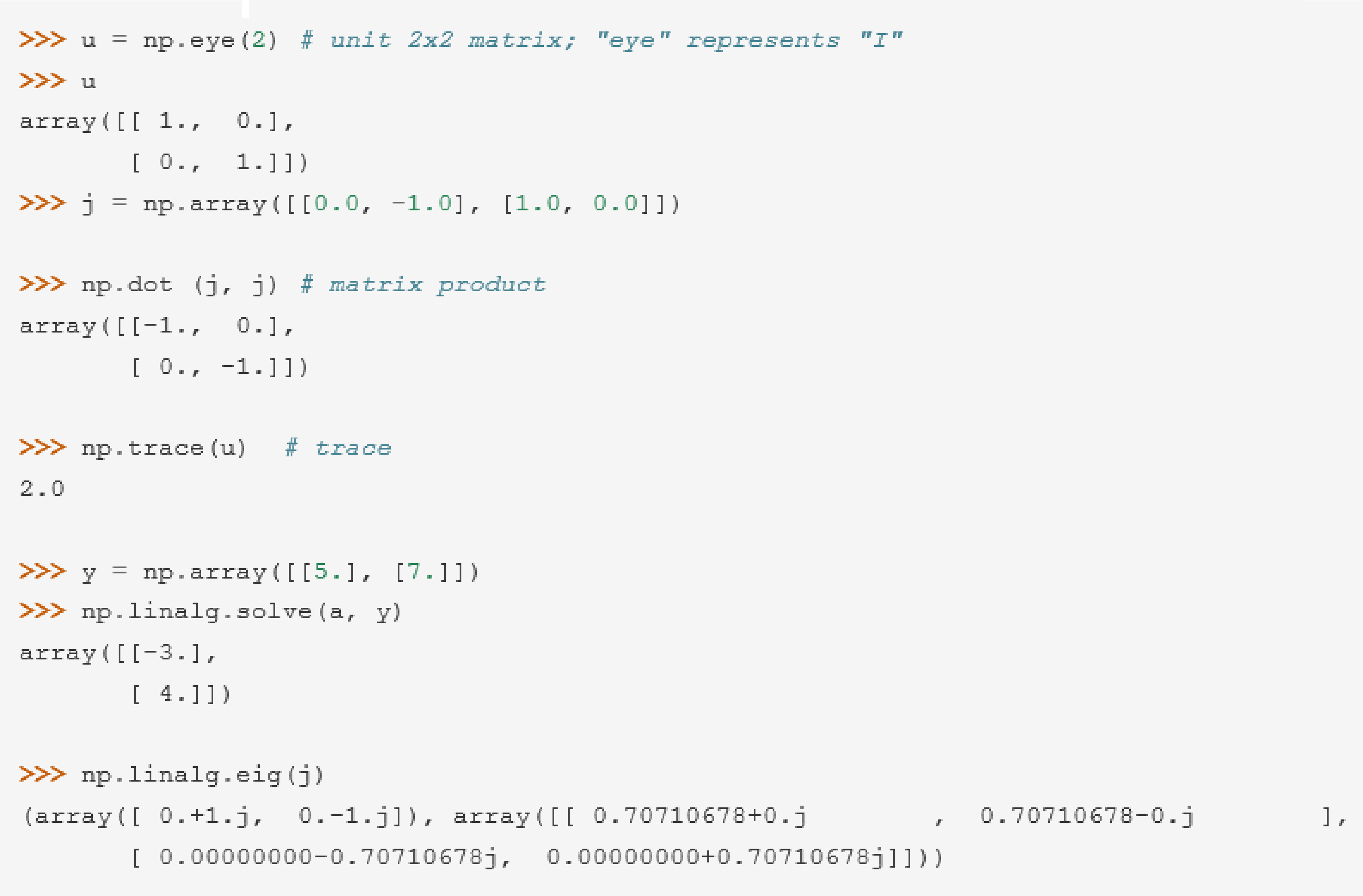
Simple Array Operations

See linalg.py in numpy folder for more.

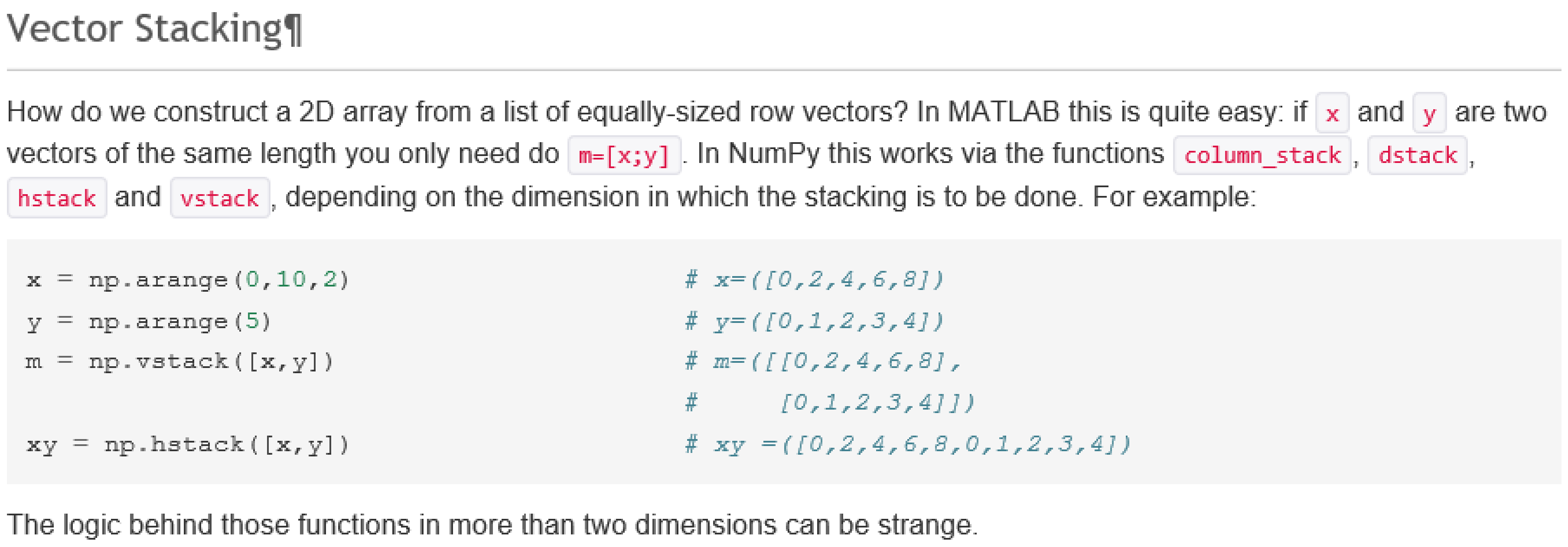
```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]

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

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



```
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]`` .
```



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

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

>>> np.trace(u)  # trace
2.0

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

>>> np.linalg.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]]))
```



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:

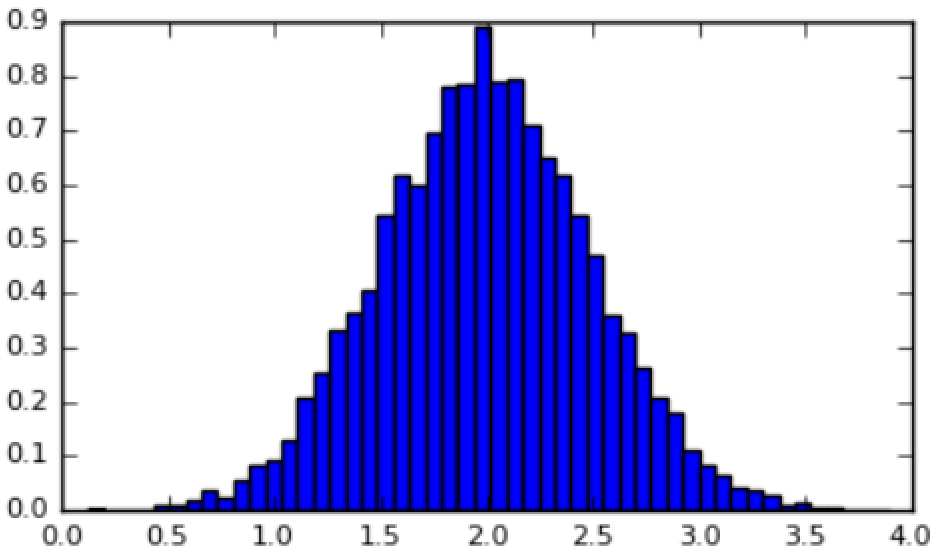
```
x = np.arange(0,10,2)                     # x=([0,2,4,6,8])
y = np.arange(5)                          # y=([0,1,2,3,4])
m = np.vstack([x,y])                      # m=([[0,2,4,6,8],
                                          #     [0,1,2,3,4]])
xy = np.hstack([x,y])                     # xy =([0,2,4,6,8,0,1,2,3,4])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

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

Histograms

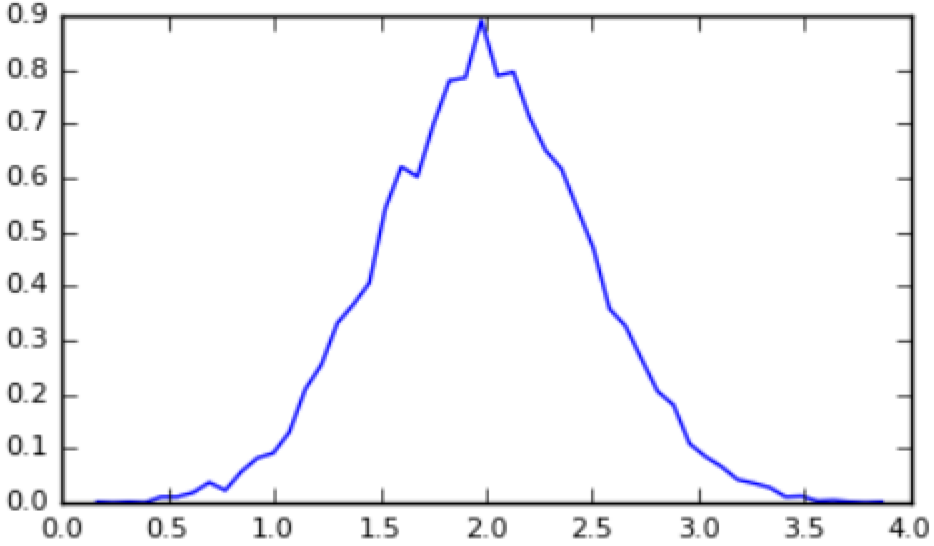
The NumPy `histogram` function applied to an array returns a pair of vectors: the histogram of the array and the vector of bins. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist` plots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

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
>>> mu, sigma = 2, 0.5
>>> v = np.random.normal(mu,sigma,10000)
>>> # Plot a normalized histogram with 50 bins
>>> plt.hist(v, bins=50, normed=1)       # matplotlib version (plot)
>>> plt.show()
```



```
>>> # Compute the histogram with numpy and then plot it
>>> (n, bins) = np.histogram(v, bins=50, normed=True)  # NumPy version (no plot)
>>> plt.plot(.5*(bins[1:]+bins[:-1]), n)
>>> plt.show()
```

(png, pdf)

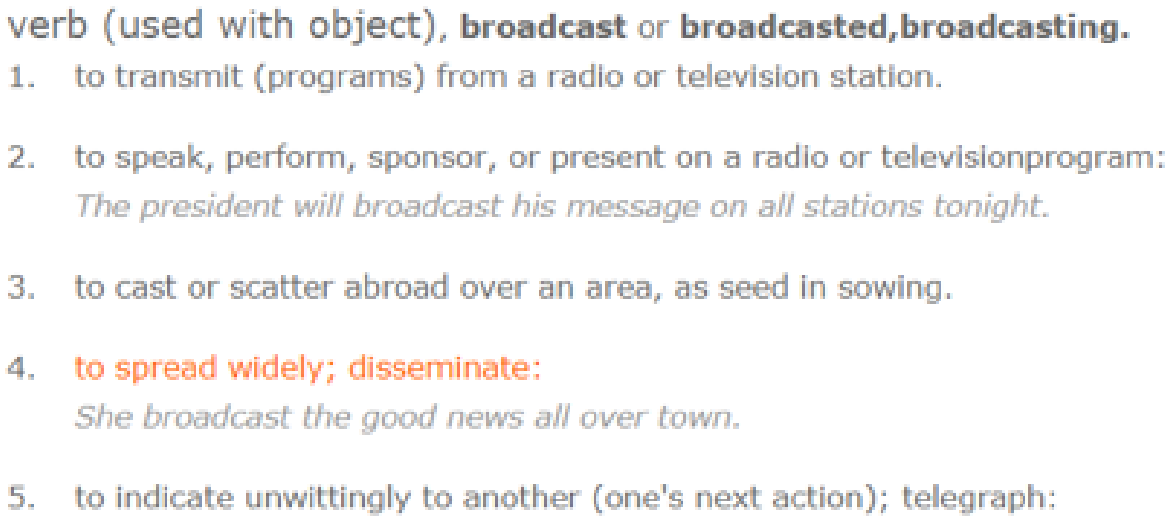


Further reading

- The Python tutorial
- NumPy Reference
- SciPy Tutorial
- SciPy Lecture Notes
- A matlab, R, IDL, NumPy/SciPy dictionary

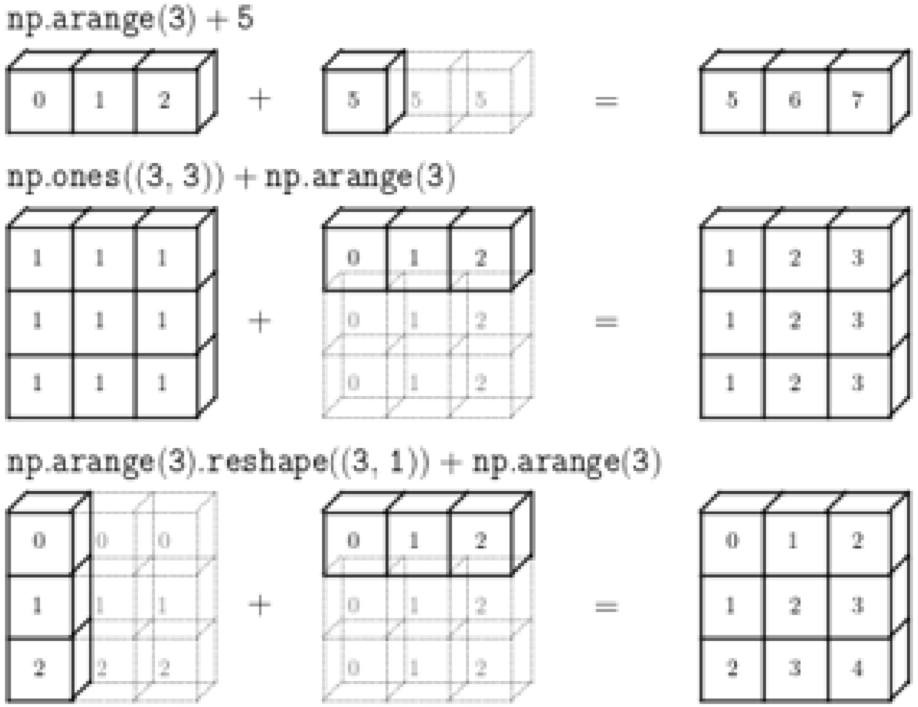
Array operation (Broadcasting)

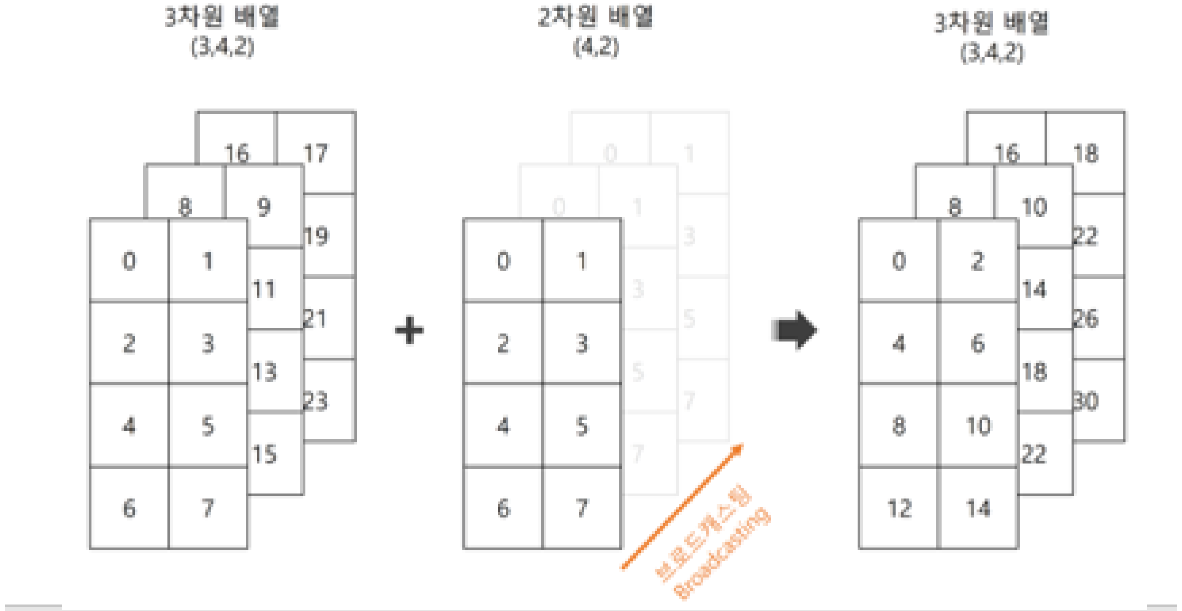
- Broadcasting(=vectorization in Matlab)



verb (used with object), **broadcast** or **broadcasted,broadcasting.**

1. to transmit (programs) from a radio or television station.
2. to speak, perform, sponsor, or present on a radio or televisionprogram:
   The president will broadcast his message on all stations tonight.
3. to cast or scatter abroad over an area, as seed in sowing.
4. to spread widely; disseminate:
   She broadcast the good news all over town.
5. to indicate unwittingly to another (one's next action); telegraph:


np.arange(3) + 5
np.ones((3, 3)) + np.arange(3)
np.arange(3).reshape((3, 1)) + np.arange(3)


3차원 배열
(3,4,2)
2차원 배열
(4,2)
3차원 배열
(3,4,2)
브로드캐스팅
Broadcasting

```
>>> a = np.arange(1, 25).reshape(4, 6)
>>> b = np.arange(25, 49).reshape(4, 6)
>>> a+b
array([[26, 28, 30, 32, 34, 36],
      [38, 40, 42, 44, 46, 48],
      [50, 52, 54, 56, 58, 60],
      [62, 64, 66, 68, 70, 72]])
```

```
>>> a = np.arange(1, 25).reshape(4, 6)
>>> a+100
array([[101, 102, 103, 104, 105, 106],
      [107, 108, 109, 110, 111, 112],
      [113, 114, 115, 116, 117, 118],
      [119, 120, 121, 122, 123, 124]])
```

```
>>> a = np.arange(5).reshape((1, 5))
>>> b = np.arange(5).reshape((5, 1))
>>> a+b
array([[0, 1, 2, 3, 4],
      [1, 2, 3, 4, 5],
      [2, 3, 4, 5, 6],
      [3, 4, 5, 6, 7],
      [4, 5, 6, 7, 8]])
```

Plotting with Matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

func=np.poly1d(np.array([1,2,3,4]).astype(float))
func1=func.deriv(m=1)
func2=func.deriv(m=2)
x=np.linspace(-10,10,30)
y=func(x)
y1=func1(x)
y2=func2(x)
plt.figure()
plt.plot(x,y,'ro',label='func')
plt.plot(x,y1,'g--',label='func1')
plt.plot(x,y2,'b--',label='func2')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.title('exercise')
plt.legend()
plt.show()
```

```
#Pyplot API를 이용하는 방식:커맨드 방식
plt.subplot(311)
plt.plot(x,y,'r-')
plt.title("Polynomial")

plt.subplot(312)
plt.plot(x,y1,'b^')
plt.title("First Derivative")

plt.subplot(313)
plt.plot(x,y,'go')
plt.title("Secod Derivative")

plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
#객체지향 API를 이용하는 방식
fig=plt.figure()
ax=fig.add_subplot(311)
ax.plot(x,y,'r-')
ax.set_title("Polynomial")

ax=fig.add_subplot(312)
ax.plot(x,y1,'b^')
ax.set_title("First Derivative")

ax=fig.add_subplot(313)
ax.plot(x,y2,'go')
ax.set_title("Second Derivative")
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

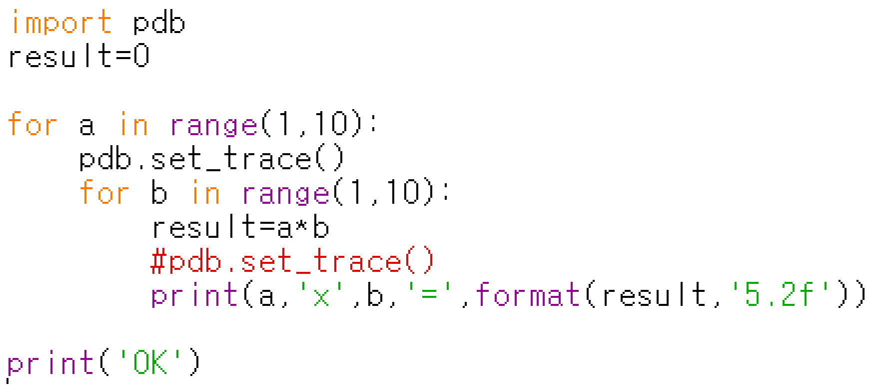
Plotting with Matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

rng=np.arange(50)
rnd=np.random.randint(0,10,size=(3,rng.size))
yrs=1950+rng

fig,ax=plt.subplots(figsize=(5,3))
ax.stackplot(yrs,rng+rnd,labels=['Eastasia','Eurisia','Oceania'])
ax.set_title('Combined debt growth over time')
ax.legend(loc='upper left')
ax.set_ylabel('Total debt')
ax.set_xlim(xmin=yrs[0],xmax=yrs[-1])
fig.tight_layout()
plt.show()
```

Debugging

컴퓨터 프로그램의 정확성이나 논리적인 오류(버그)를 찾아내는 테스트 과정
-step over/step into
-중단점(breakpoint) 설정
-콜스택 검사
-소스 리스팅
-변수치환 등 다양한 기능

파이썬의 디버깅
- Pdb 모듈 사용



```python
import pdb
result=0

for a in range(1,10):
    pdb.set_trace()
    for b in range(1,10):
        result=a*b
        #pdb.set_trace()
        print(a,'x',b,'=',format(result,'5.2f'))

print('OK')
```

비교기능:
(pdb)c
(pdb)n
(pdb)print(변수명)

PDB 명령어

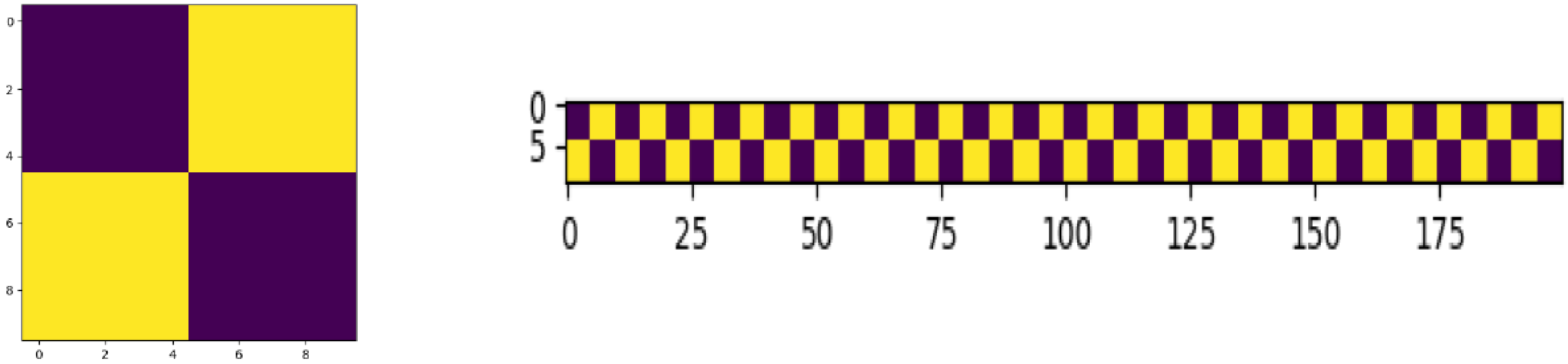
- https://docs.python.org/3/library/pdb.html

PDB 명령어	실행내용
help	도움말
next	다음 문장으로 이동
print	변수값 화면에 표시
list	소스코드 리스트 출력. 현재 위치 화살표로 표시됨
where	콜스택 출력
continue	계속 실행. 다음 중단점에 멈추거나 중단점 없으면 끝까지 실행
step	Step Into 하여 함수 내부로 들어감
return	현재 함수의 리턴 직전까지 실행
!변수명 = 값	변수에 값 재설정

Numpy/matplotlib 연습

1

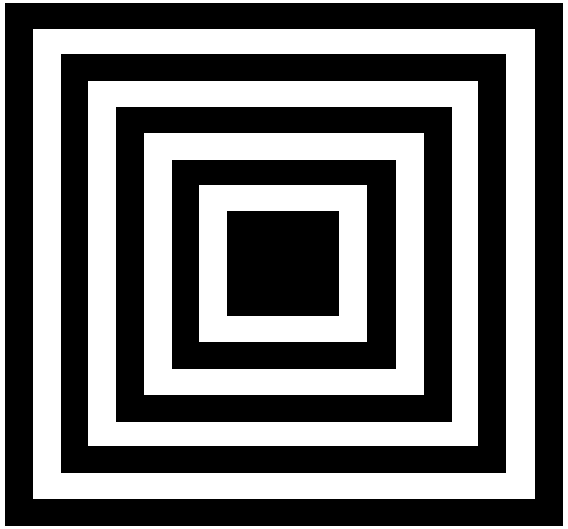
5X5 크기를 갖는 empty, full 변수를 선언 후 empty 에는 0을, full 에는 255를 채운다. 이 두 변수를 이용하여 아래 그림처럼 10x10 크기의 행렬을 만든 후 이를 imsi1으로 저장하라. imsi1을 수평으로 붙여나가 10x200 크기의 행렬을 imsi2로 저장하고 출력하라.



Numpy/matplotlib 연습

2

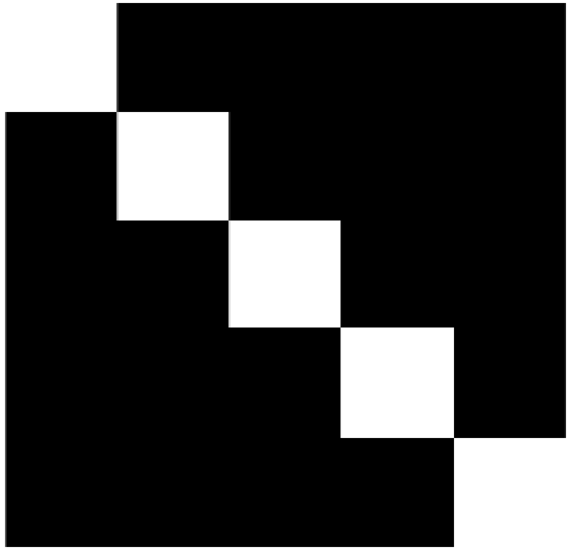
200x200의 imsi3을 만든 후 좌, 우, 위, 아래가 각 10씩 줄어들며 색이 바뀌어 입력되는 아래와 같은 결과를 출력하라. 역시 for 문 사용할 것.



Numpy/matplotlib 연습

3

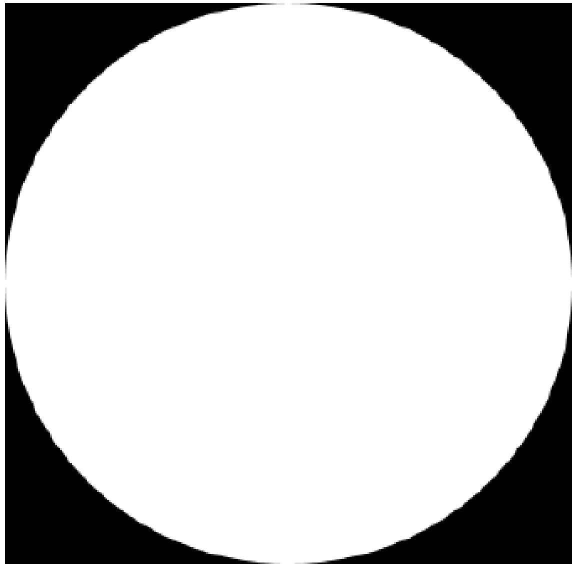
200x200의 imsi4를 만든 후 아래 그림처럼 크기 40x40의 사각형이 대각선으로 출력되게 imsi5를 변형하여 출력하라



Numpy/matplotlib 연습

4

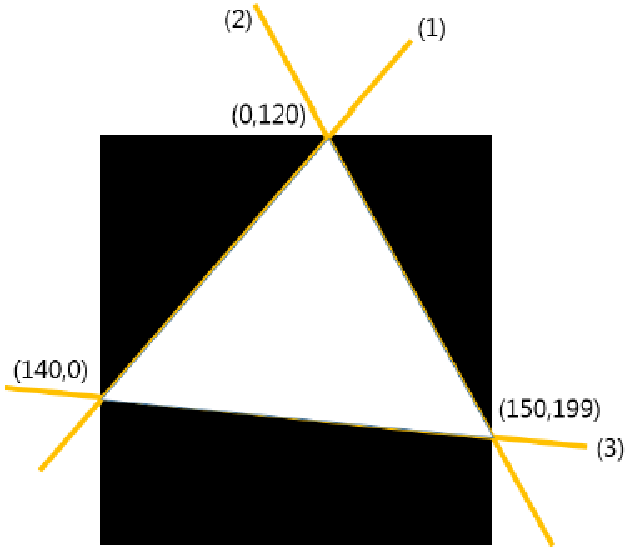
200x200의 imsi5를 만든 후 아래 그림처럼 100픽셀을 반지름으로 하는 원의 내부를 채워서 imsi5로 변형하여 출력하라. np.meshgrid를 사용할 것.



Numpy/matplotlib 연습

5

200x200의 imsi6을 만든 후, 아래 그림에서 주어진 3 교점으로 직선 (1),(2),(3)을 구한 후 내부를 흰색으로 채우고 출력하라. np.meshgrid를 사용할 것.



6 윈도우 OS가 제공하는 그림판 mspaint.exe를 실행시킨 후 그림 캔버스의 크기를 800x600으로 설정하라. 이후 그림판의 연필을 선택하고 연필의 두께를 적절히 조정한 후, 역시 본인의 이름을 마우스로 쓴 다음 파일을 name.png로 저장하라. 그리고 그레이스케일로 변환하라. Matplotlib를 이용하여 그림을 출력하라.(만약 읽어들이지 못한다면 PyPI를 이용하여 pillow 라이브러리를 설치후 재시도)

7 위에서 읽어들인 그림 중 검은 색으로 표시된 픽셀 수를 카운트하라.

8 matplotlib를 사용하여 python에서 lena.bmp를 읽어 imsi8에 저장 후 imshow()를 사용하여 출력하라.

9 앞 사진을 plot_surface()을 사용하여 3D로 출력하라