Numpy基础-1 C08

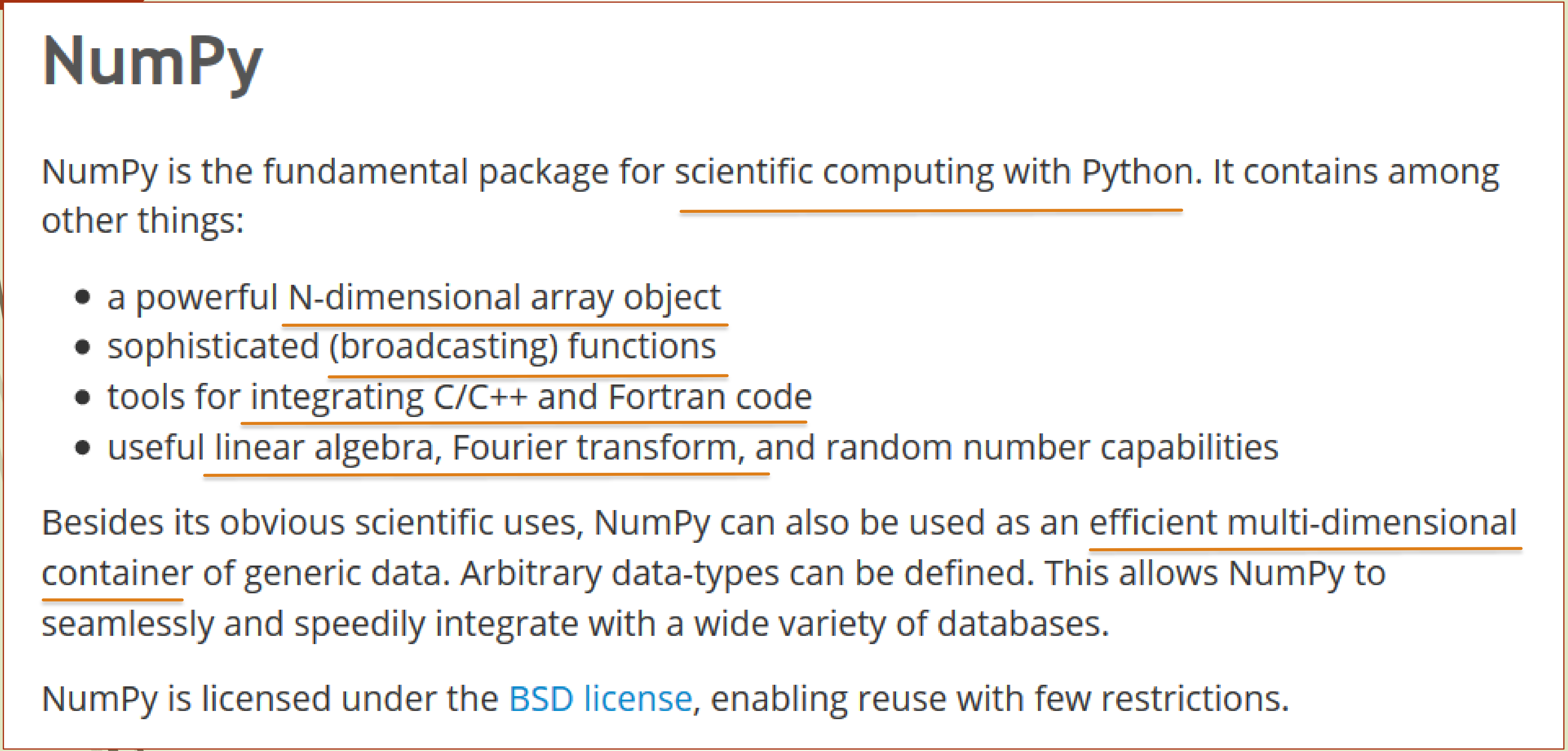
胡俊峰 北京大学

2023/03/27

本次课内容

- ndarray对象及基本操作
- Numpy矩阵运算基础
- 数据可视化的软件包：Matplotlib
- 向量计算
- K-means聚类算法
- 距离与协方差
- 矩阵的特征值与特征向量

Numpy定位



NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

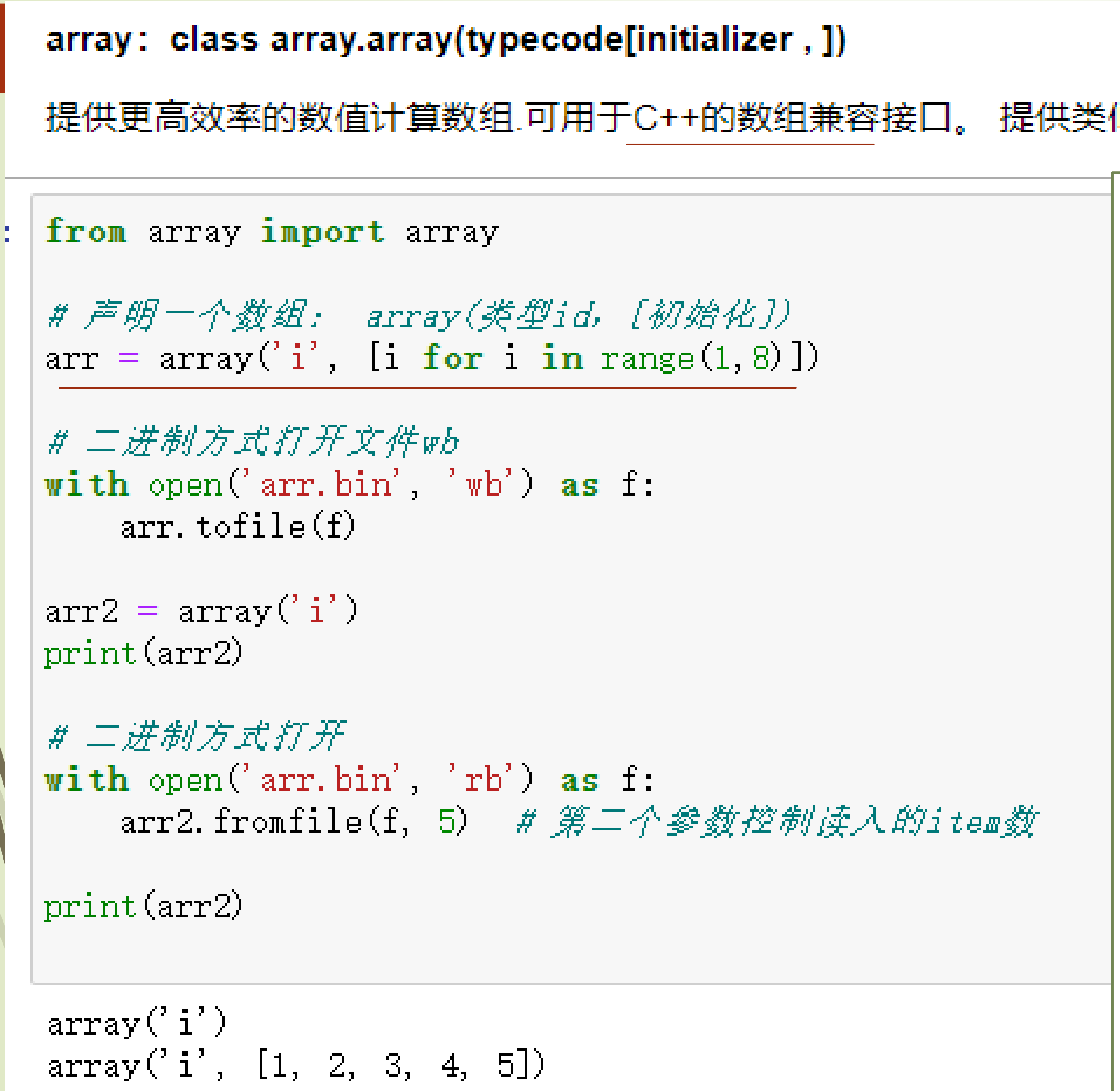
Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is licensed under the BSD license, enabling reuse with few restrictions.

Numpy中的ndarray类型

- Python中的array
- ndarray的类型、访问、切片与shape转换
- ndarray内置计算

Python中的array --- 高效的数值数组

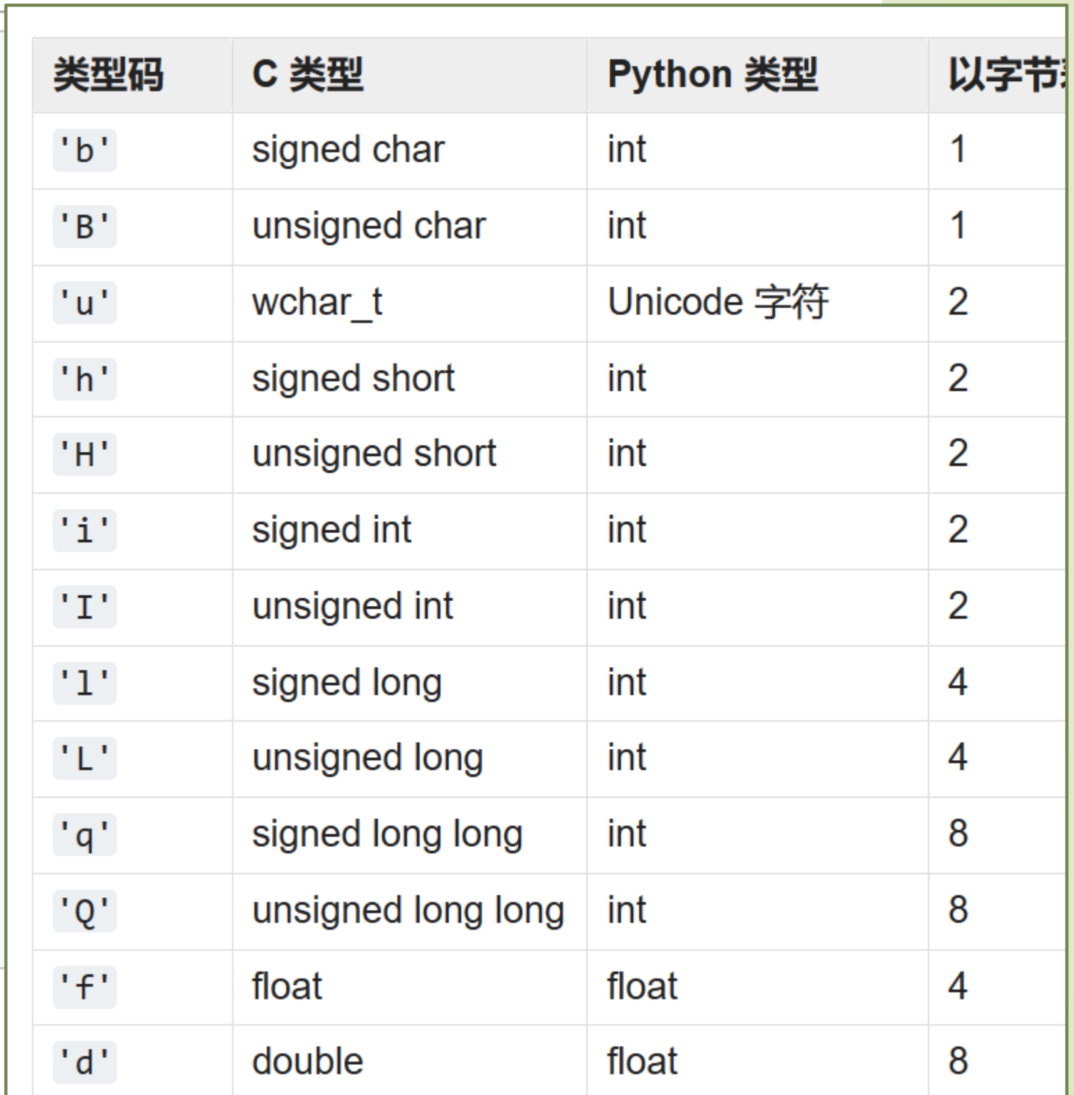


array： class array.array(typecode[initializer ,])

提供更高效率的数值计算数组.可用于C++的数组兼容接口。 提供类似list的访问和增删改操作

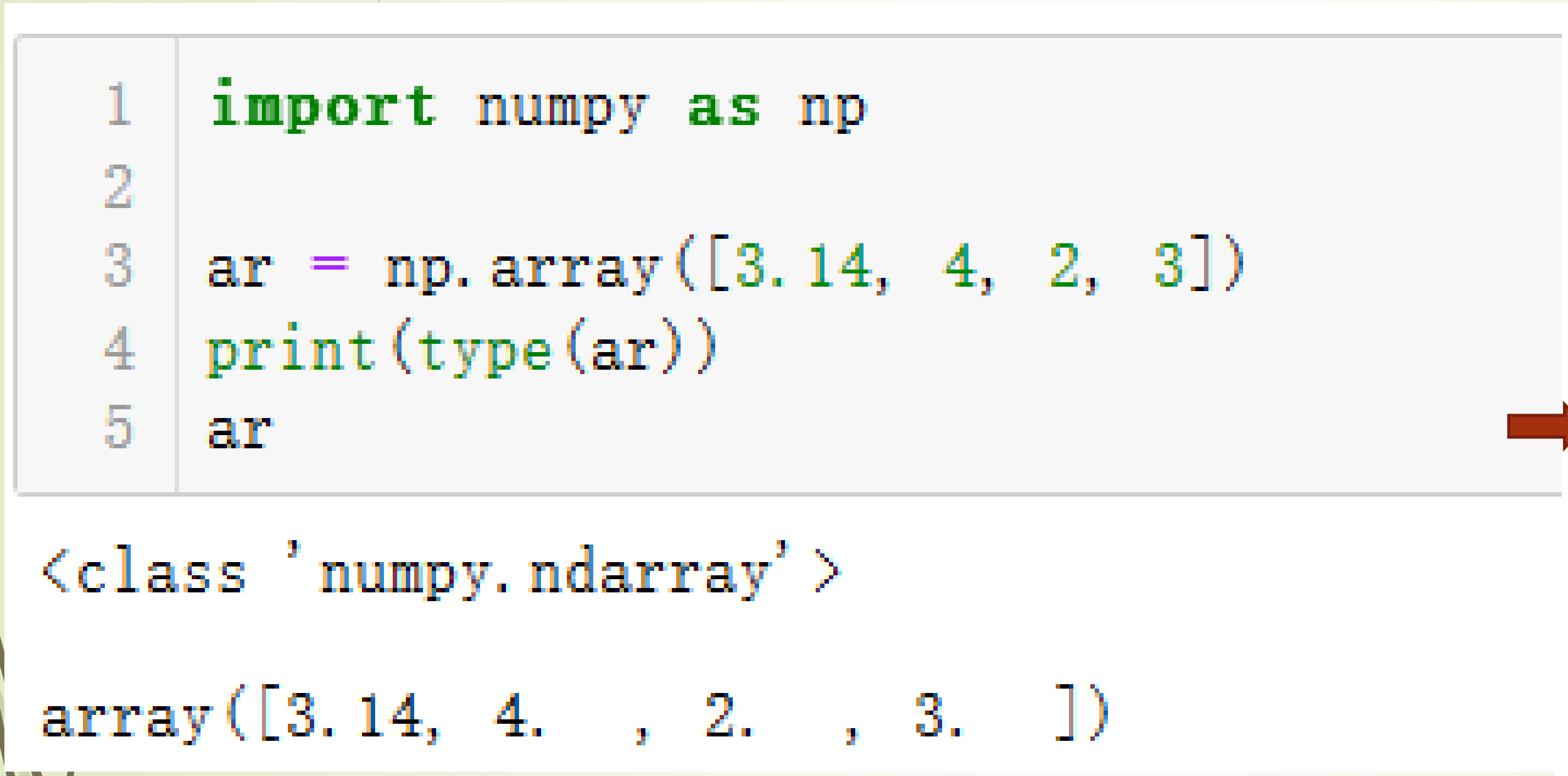
```python
from array import array

# 声明一个数组:  array(类型id, [初始化])
arr = array('i', [i for i in range(1,8)])

# 二进制方式打开文件wb
with open('arr.bin', 'wb') as f:
    arr.tofile(f)

arr2 = array('i')
print(arr2)

# 二进制方式打开
with open('arr.bin', 'rb') as f:
    arr2.fromfile(f, 5)   # 第二个参数控制读入的item数

print(arr2)
```

```
array('i')
array('i', [1, 2, 3, 4, 5])
```



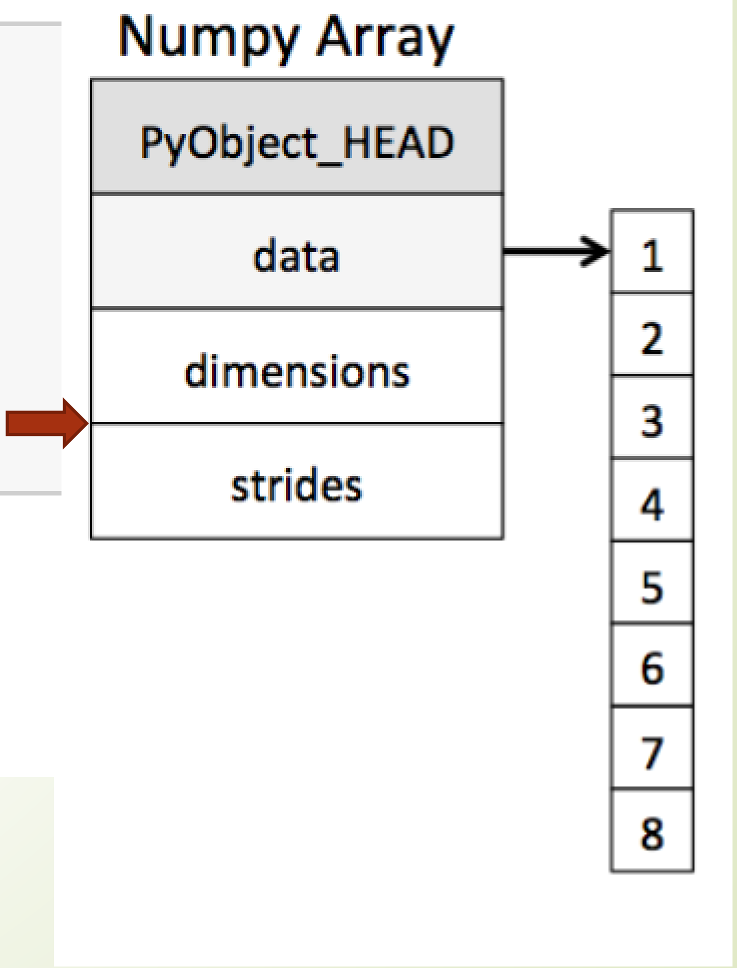
类型码	C 类型	Python 类型	以字节
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	wchar_t	Unicode 字符	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

高效的固定类型的矩阵数据处理方案

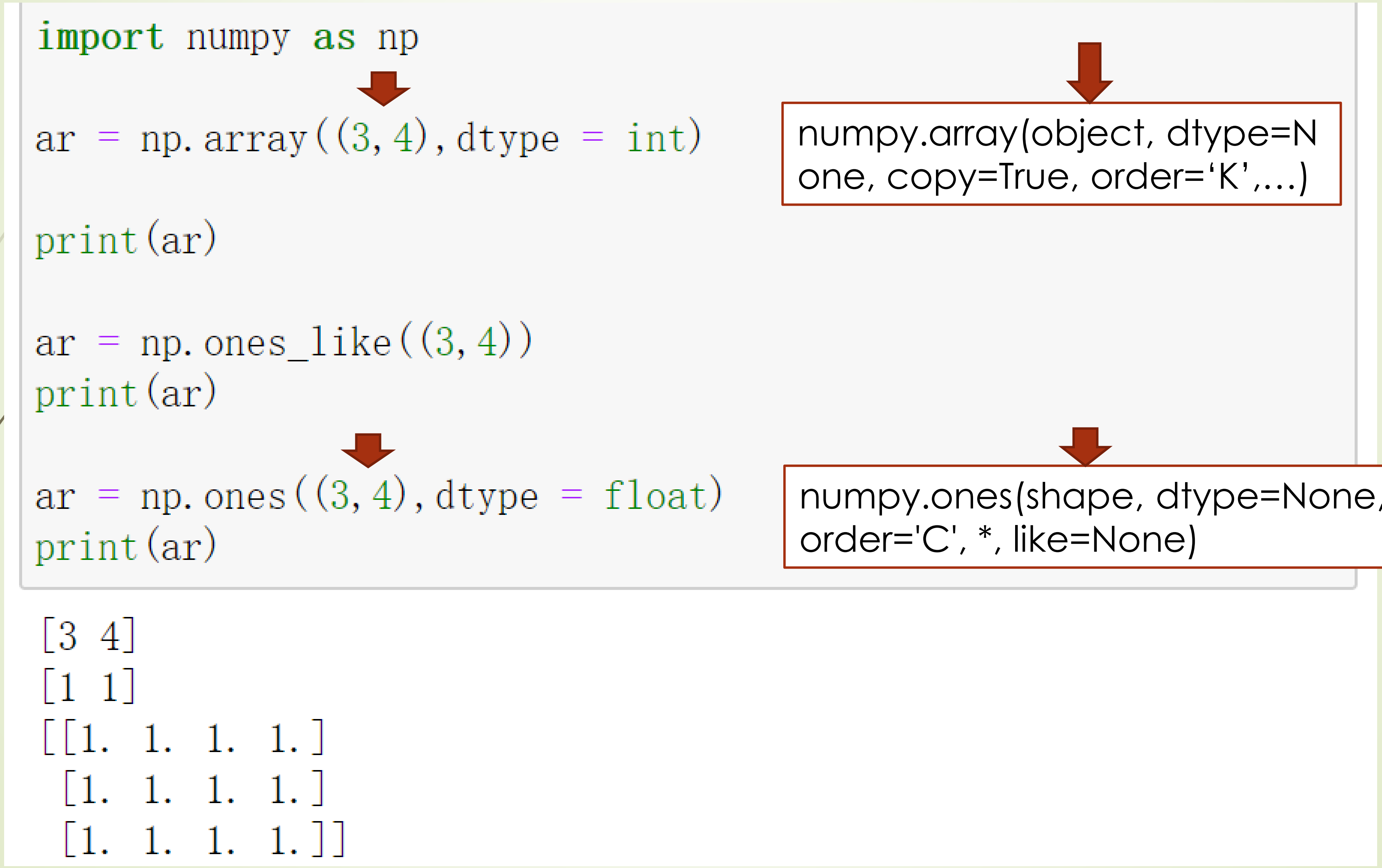


```python
import numpy as np

ar = np.array([3.14, 4, 2, 3])
print(type(ar))
ar
```

```
<class 'numpy.ndarray'>

array([3.14, 4.  , 2.  , 3.  ])
```



声明/初始化 一个ndarray对象



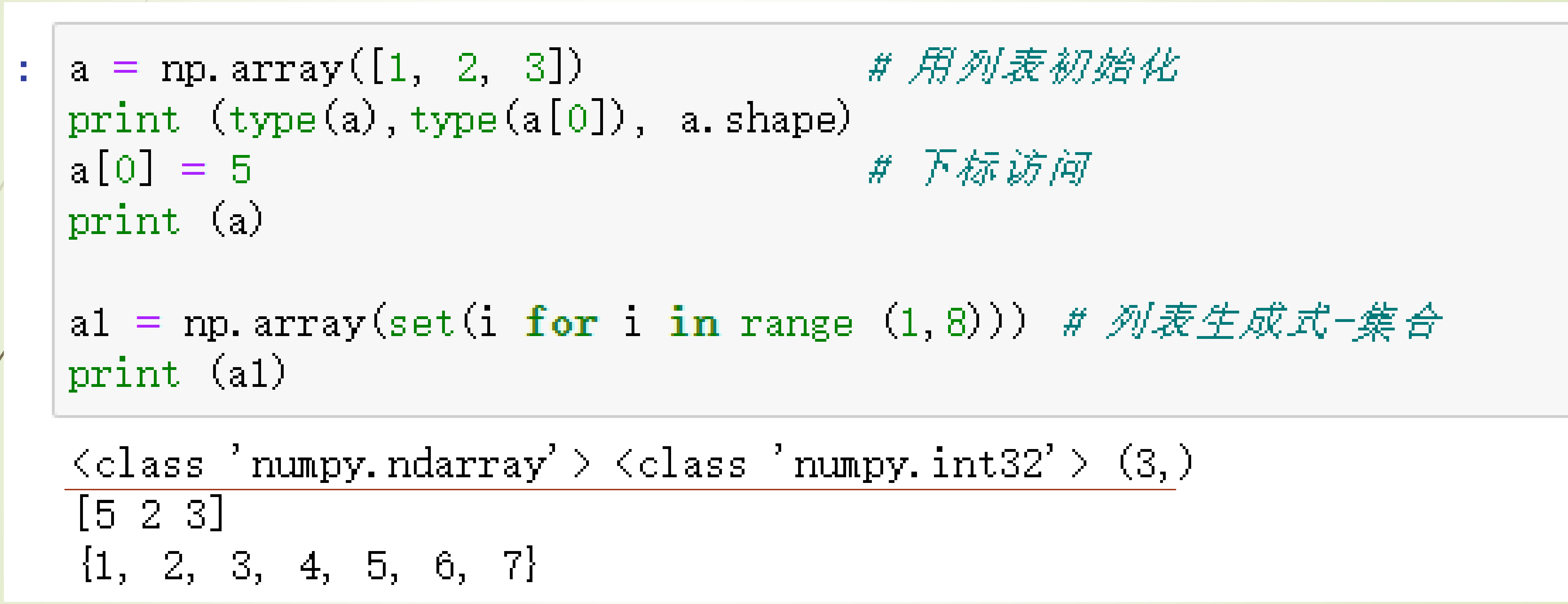
```
import numpy as np

ar = np.array((3,4),dtype = int)

print(ar)

ar = np.ones_like((3,4))
print(ar)

ar = np.ones((3,4),dtype = float)
print(ar)
```

```
[3 4]
[1 1]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

numpy.array(object, dtype=None, copy=True, order='K',...)

numpy.ones(shape, dtype=None, order='C', *, like=None)

用python的容器（迭代器）类型来初始化ndarray



```
a = np.array([1, 2, 3])                # 用列表初始化
print (type(a),type(a[0]), a.shape)
a[0] = 5                               # 下标访问
print (a)

a1 = np.array(set(i for i in range (1,8))) # 列表生成式-集合
print (a1)
```

```
<class 'numpy.ndarray'> <class 'numpy.int32'> (3,)
[5 2 3]
{1, 2, 3, 4, 5, 6, 7}
```

```
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print (b)
```

```
[[1 2 3]
 [4 5 6]]
```

一些常见的初始化矩阵方法

```
np.ones( (2,3,4), dtype=np.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)
```

```
np.empty( (2,3) )         # uninitialized, output may vary np.zeros((3,4))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

一些常见的初始化矩阵方法（续）

```python
c = np.full((2,2), 7) # Create a constant array
print (c)
```

```
[[7 7]
 [7 7]]
```

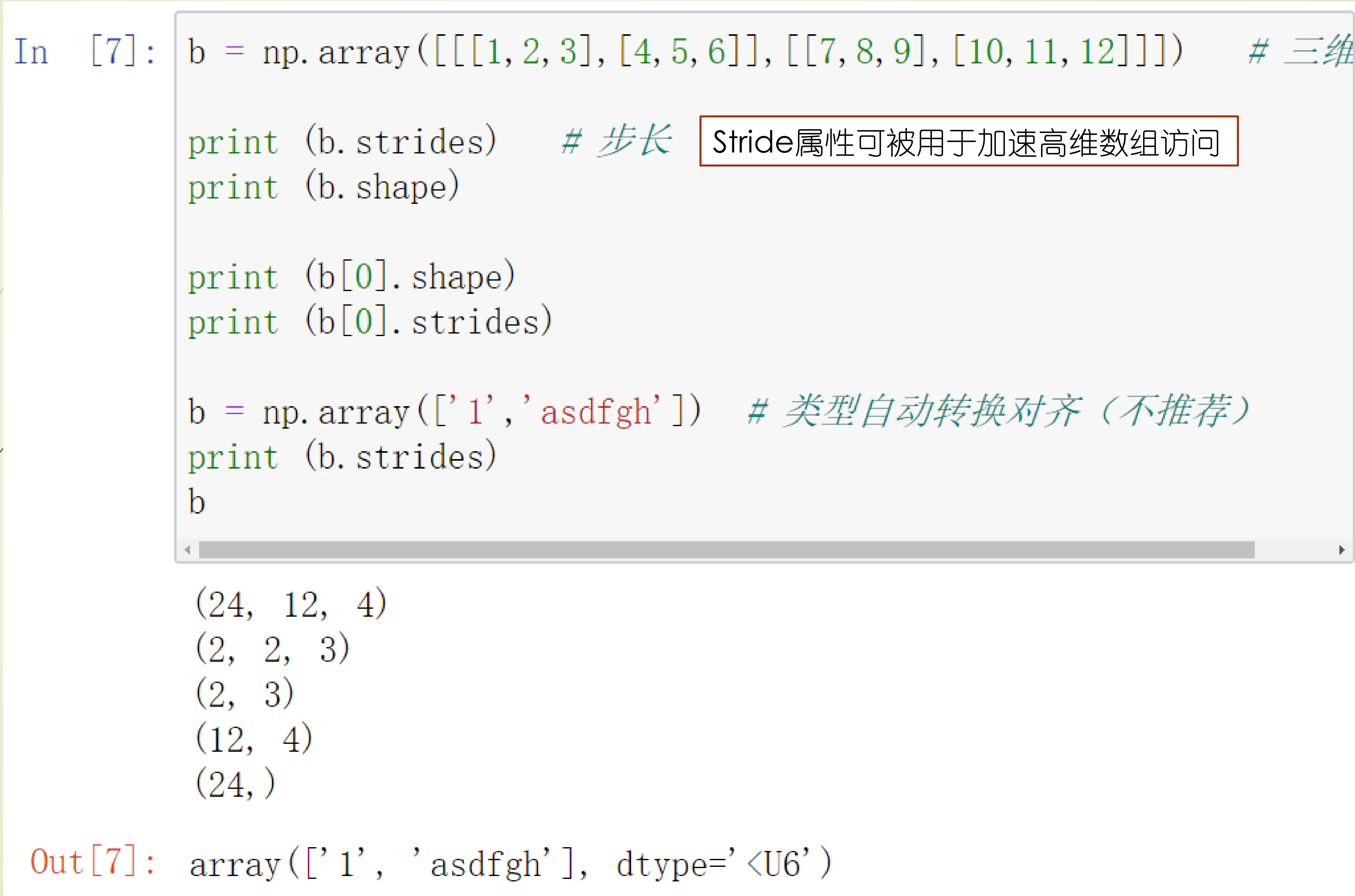
```python
d = np.eye(3)          # Create a 3x3 identity matrix
print (d)
```

```
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

```python
e = np.random.random((2,2)) # Create an array filled with random values
print (e)
```

```
[[0.90184008 0.19514327]
 [0.8609894  0.80850789]]
```

高维数组：

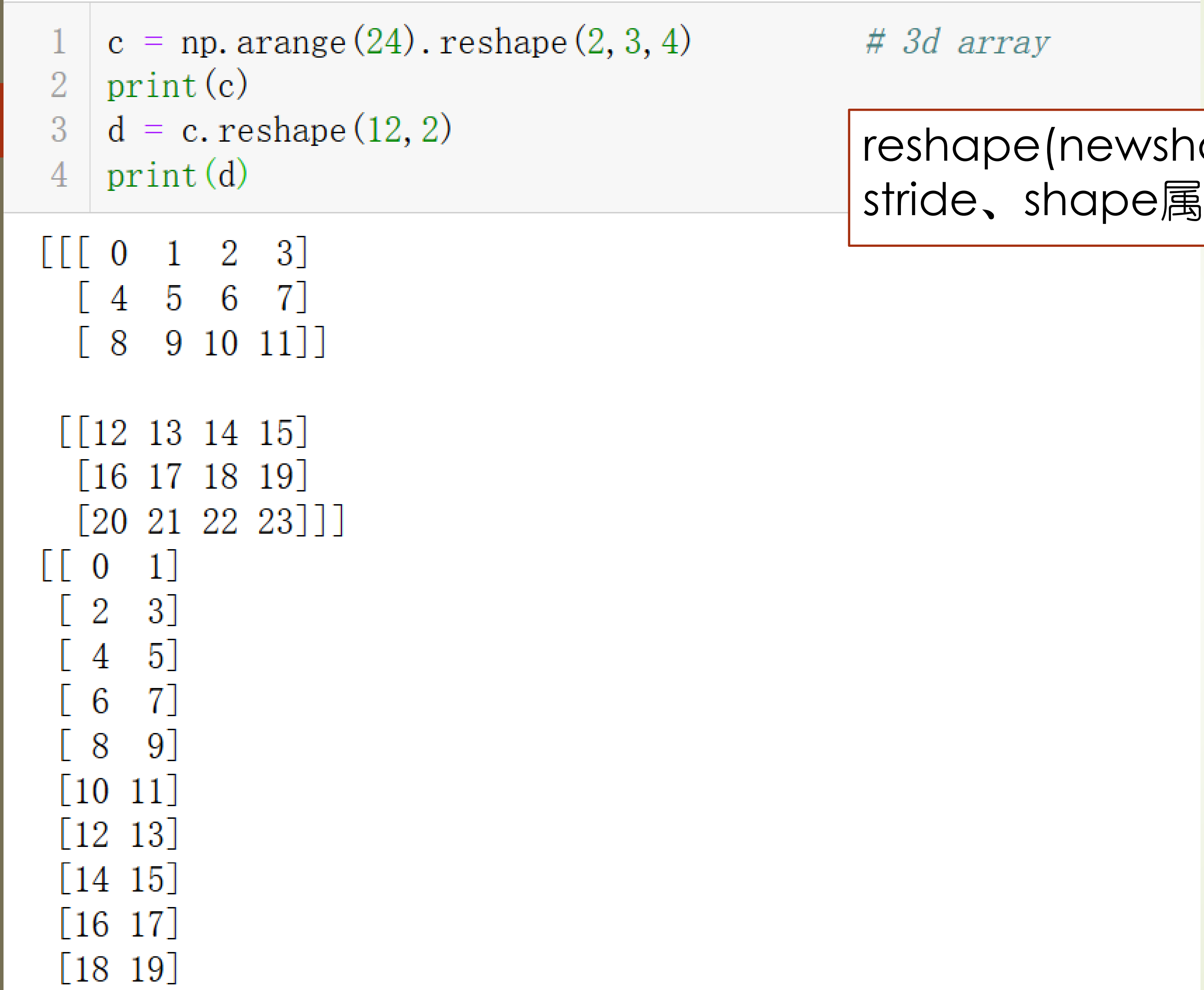


```
In  [7]: b = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])    # 三维

         print (b.strides)    # 步长
         print (b.shape)

         print (b[0].shape)
         print (b[0].strides)

         b = np.array(['1','asdfgh'])   # 类型自动转换对齐（不推荐）
         print (b.strides)
         b
```

Stride属性可被用于加速高维数组访问

```
(24, 12, 4)
(2, 2, 3)
(2, 3)
(12, 4)
(24,)
```

```
Out[7]: array(['1', 'asdfgh'], dtype='<U6')
```

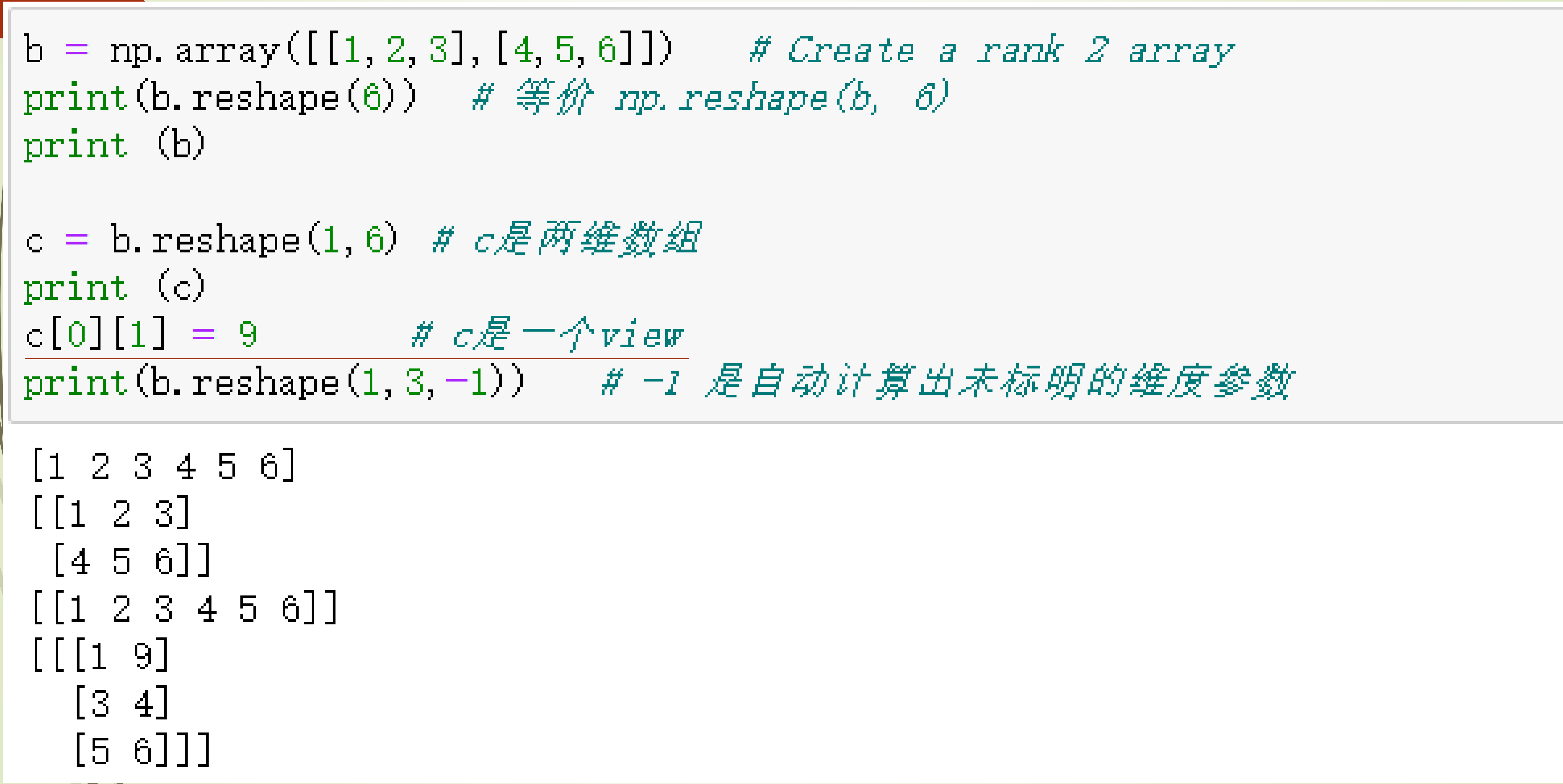


```python
c = np.arange(24).reshape(2,3,4)          # 3d array
print(c)
d = c.reshape(12,2)
print(d)
```

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

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]
 [12 13]
 [14 15]
 [16 17]
 [18 19]
```

reshape(newshape),like(b)
stride、shape属性调整，数据不变

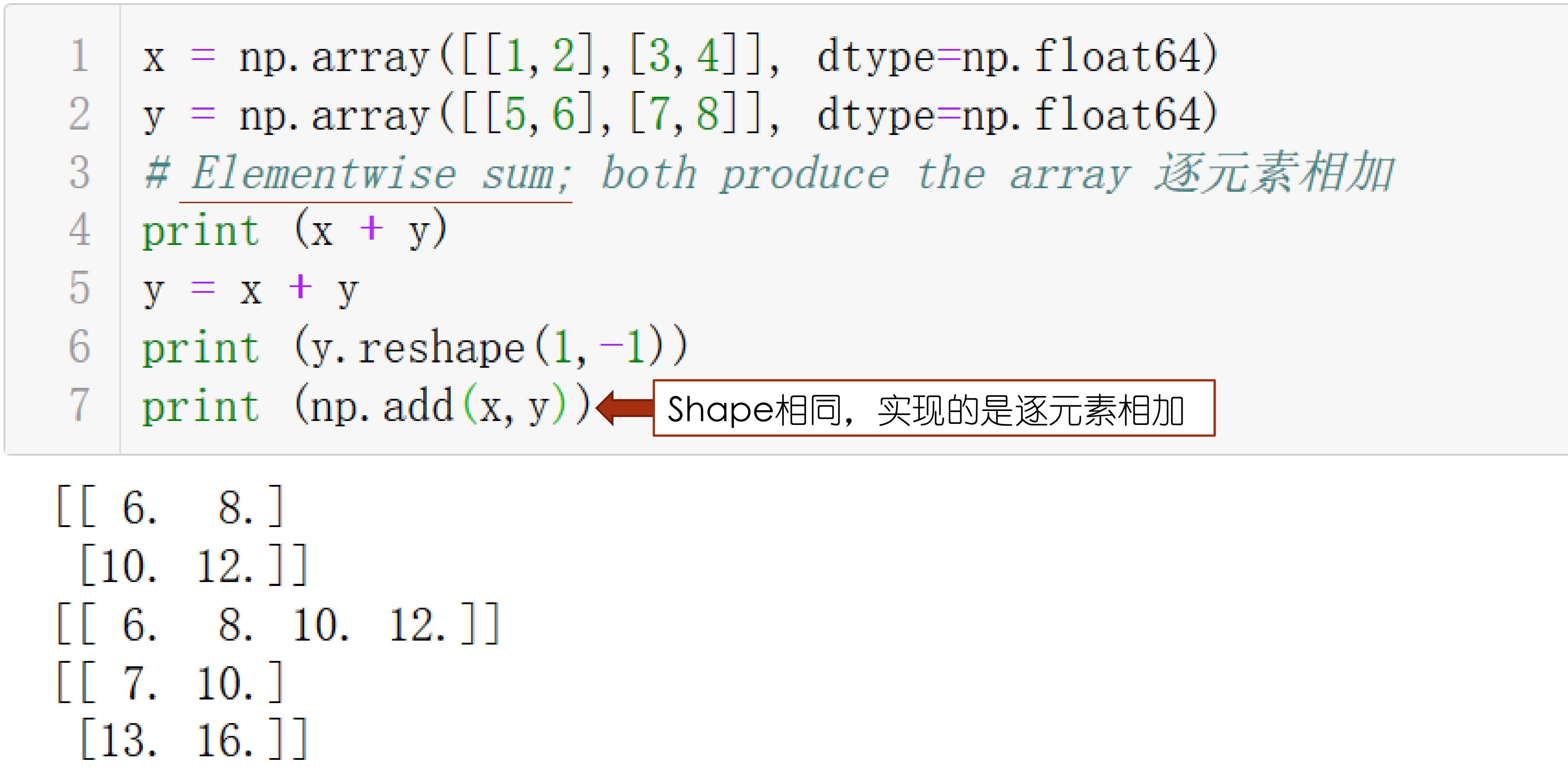
Reshape例子：



```
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.reshape(6))   # 等价 np.reshape(b, 6)
print (b)

c = b.reshape(1,6) # c是两维数组
print (c)
c[0][1] = 9         # c是一个view
print(b.reshape(1,3,-1))    # -1 是自动计算出未标明的维度参数
```

```
[1 2 3 4 5 6]
[[1 2 3]
 [4 5 6]]
[[1 2 3 4 5 6]]
[[[1 9]
  [3 4]
  [5 6]]]
```

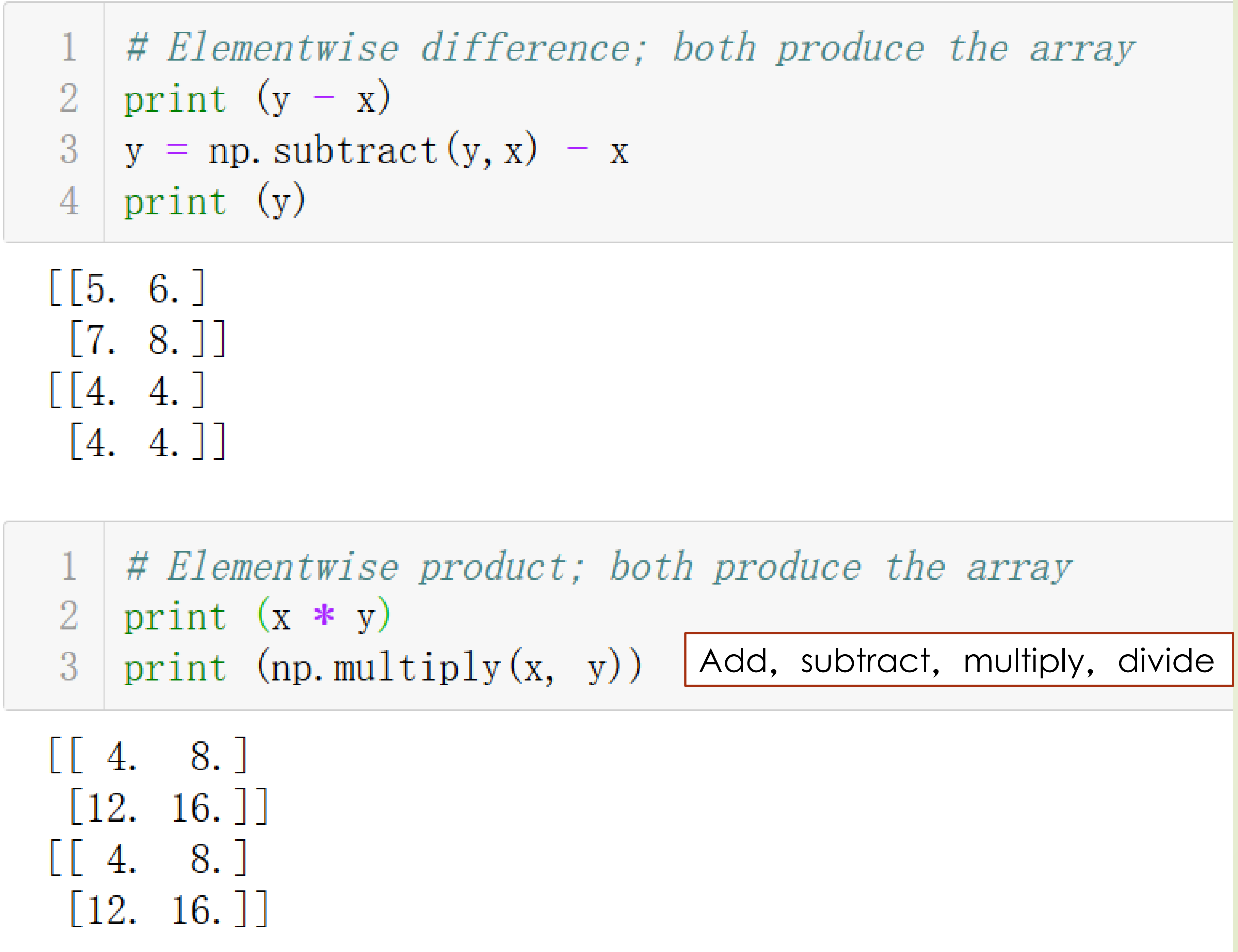
Array math (数组间的算术运算，逐元素对应计算)



```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
# Elementwise sum; both produce the array 逐元素相加
print (x + y)
y = x + y
print (y.reshape(1,-1))
print (np.add(x,y))
```

Shape相同，实现的是逐元素相加

```
[[ 6.   8.]
 [10.  12.]]
[[ 6.   8.  10.  12.]]
[[ 7.  10.]
 [13.  16.]]
```



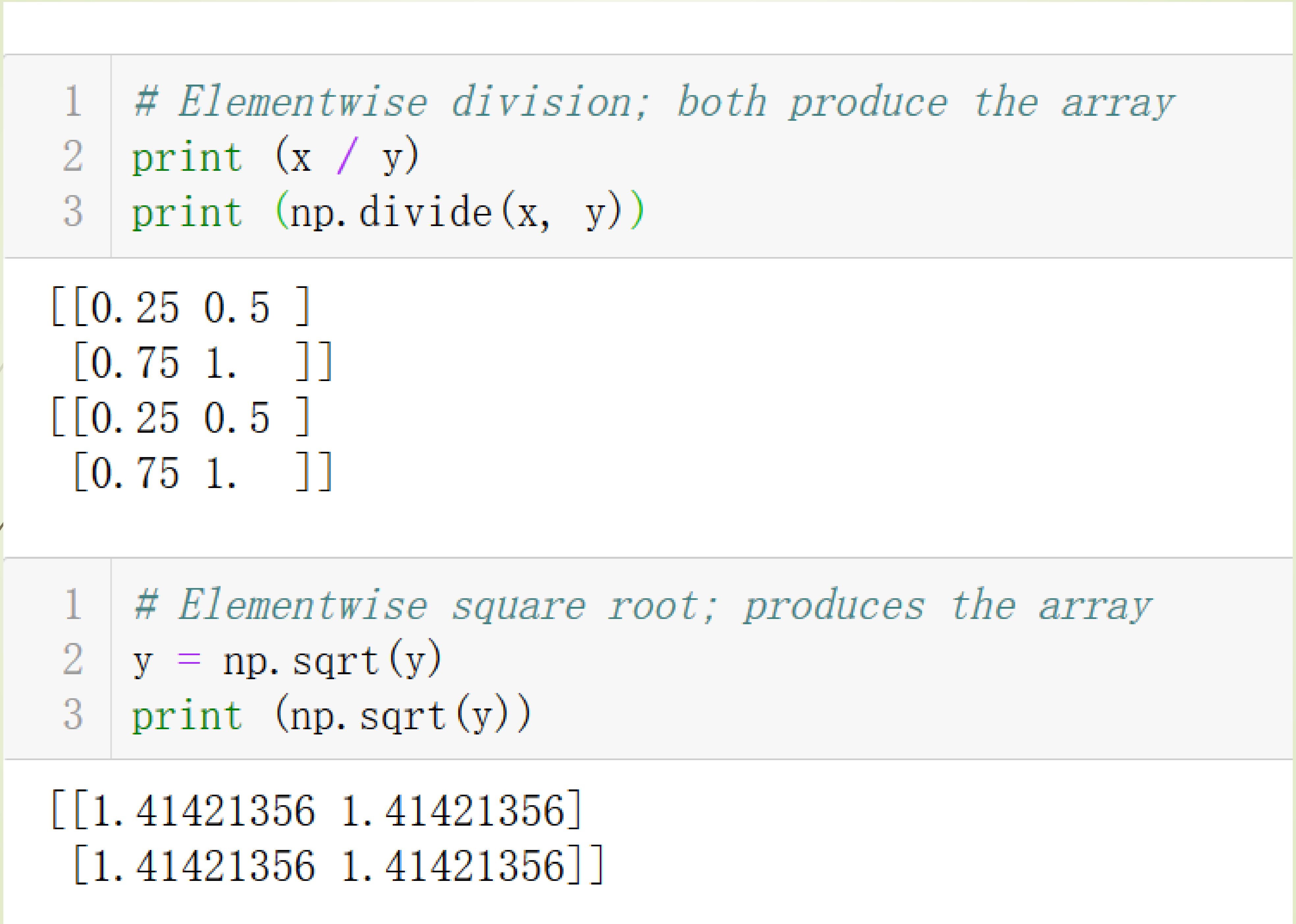
```python
# Elementwise difference; both produce the array
print (y - x)
y = np.subtract(y,x) - x
print (y)
```

```
[[5.  6.]
 [7.  8.]]
[[4.  4.]
 [4.  4.]]
```

```python
# Elementwise product; both produce the array
print (x * y)
print (np.multiply(x,  y))
```

Add, subtract, multiply, divide

```
[[ 4.   8.]
 [12.  16.]]
[[ 4.   8.]
 [12.  16.]]
```



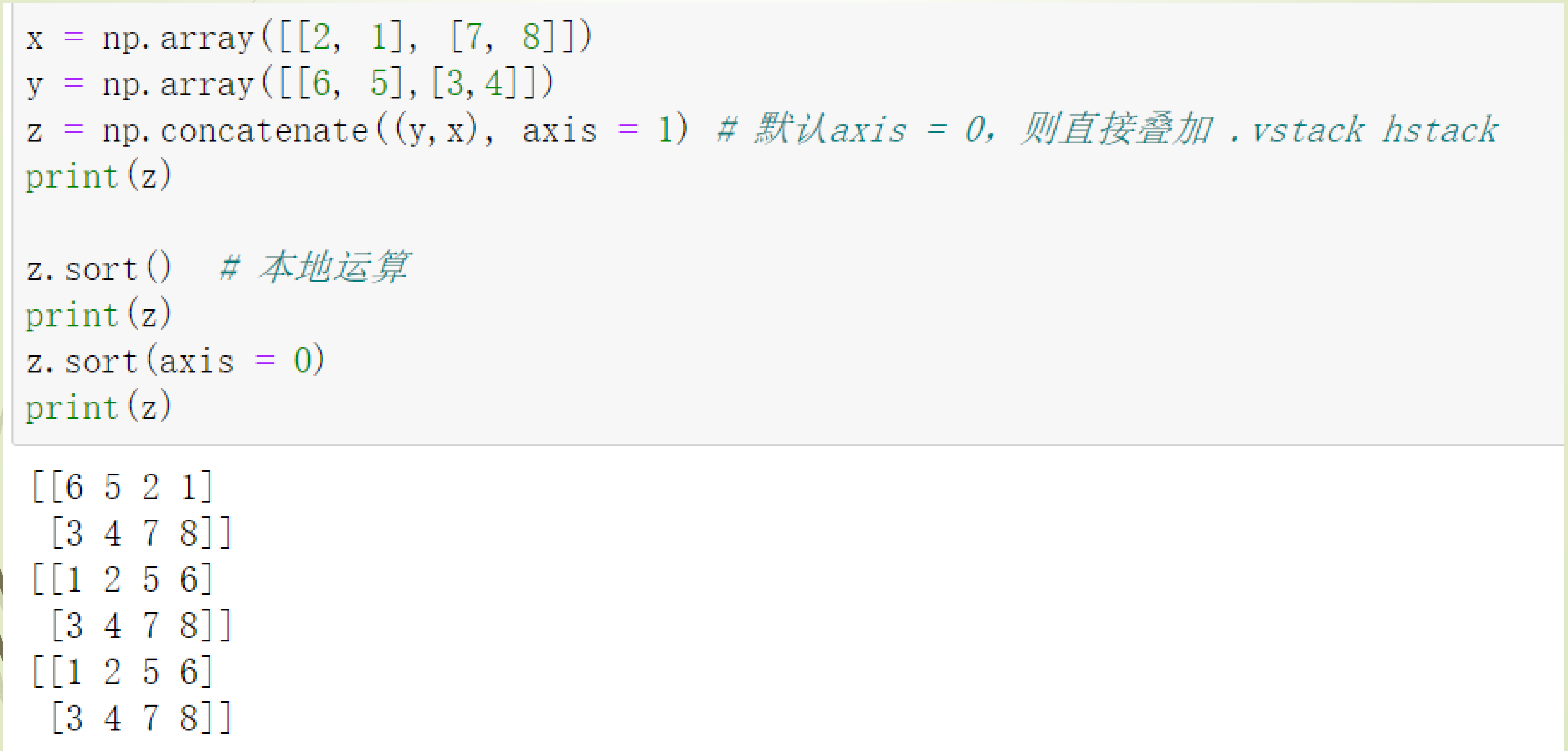
```python
# Elementwise division; both produce the array
print (x / y)
print (np.divide(x, y))
```

```
[[0.25 0.5 ]
 [0.75 1.  ]]
[[0.25 0.5 ]
 [0.75 1.  ]]
```

```python
# Elementwise square root; produces the array
y = np.sqrt(y)
print (np.sqrt(y))
```

```
[[1.41421356 1.41421356]
 [1.41421356 1.41421356]]
```

ndarray的sort()方法



```
x = np.array([[2,  1],  [7,  8]])
y = np.array([[6,  5],[3,4]])
z = np.concatenate((y,x), axis = 1) # 默认axis = 0，则直接叠加 .vstack hstack
print(z)

z.sort()  # 本地运算
print(z)
z.sort(axis = 0)
print(z)
```

```
[[6 5 2 1]
 [3 4 7 8]]
[[1 2 5 6]
 [3 4 7 8]]
[[1 2 5 6]
 [3 4 7 8]]
```

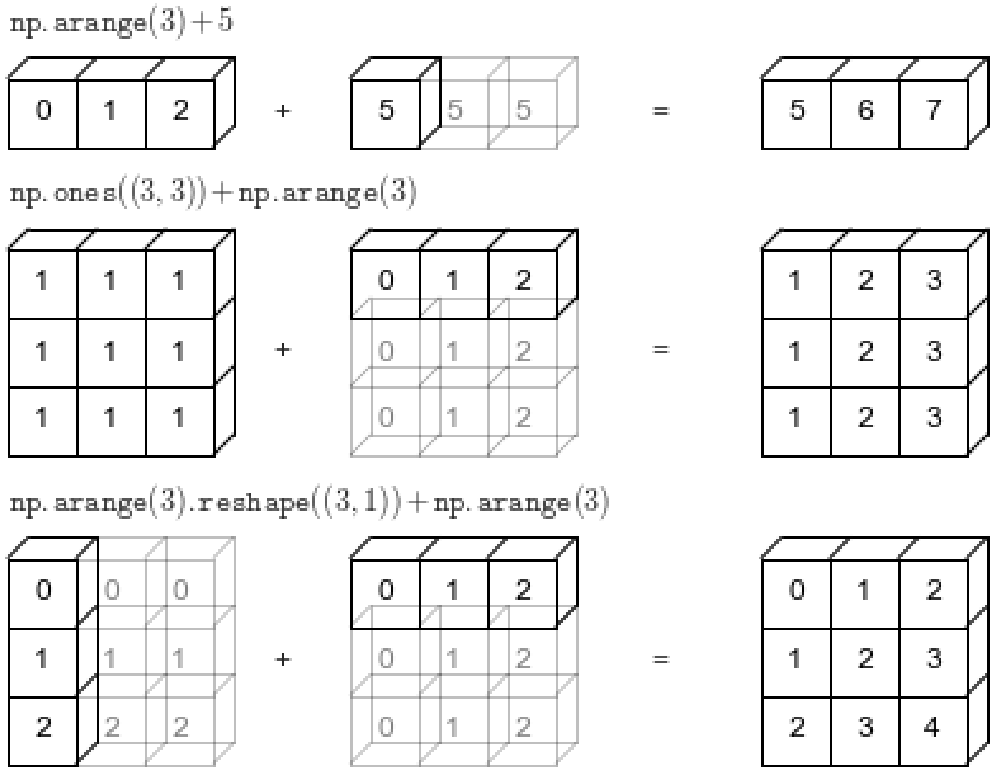
Broadcasting

Numpy中 常量-向量-矩阵运算中的广播机制

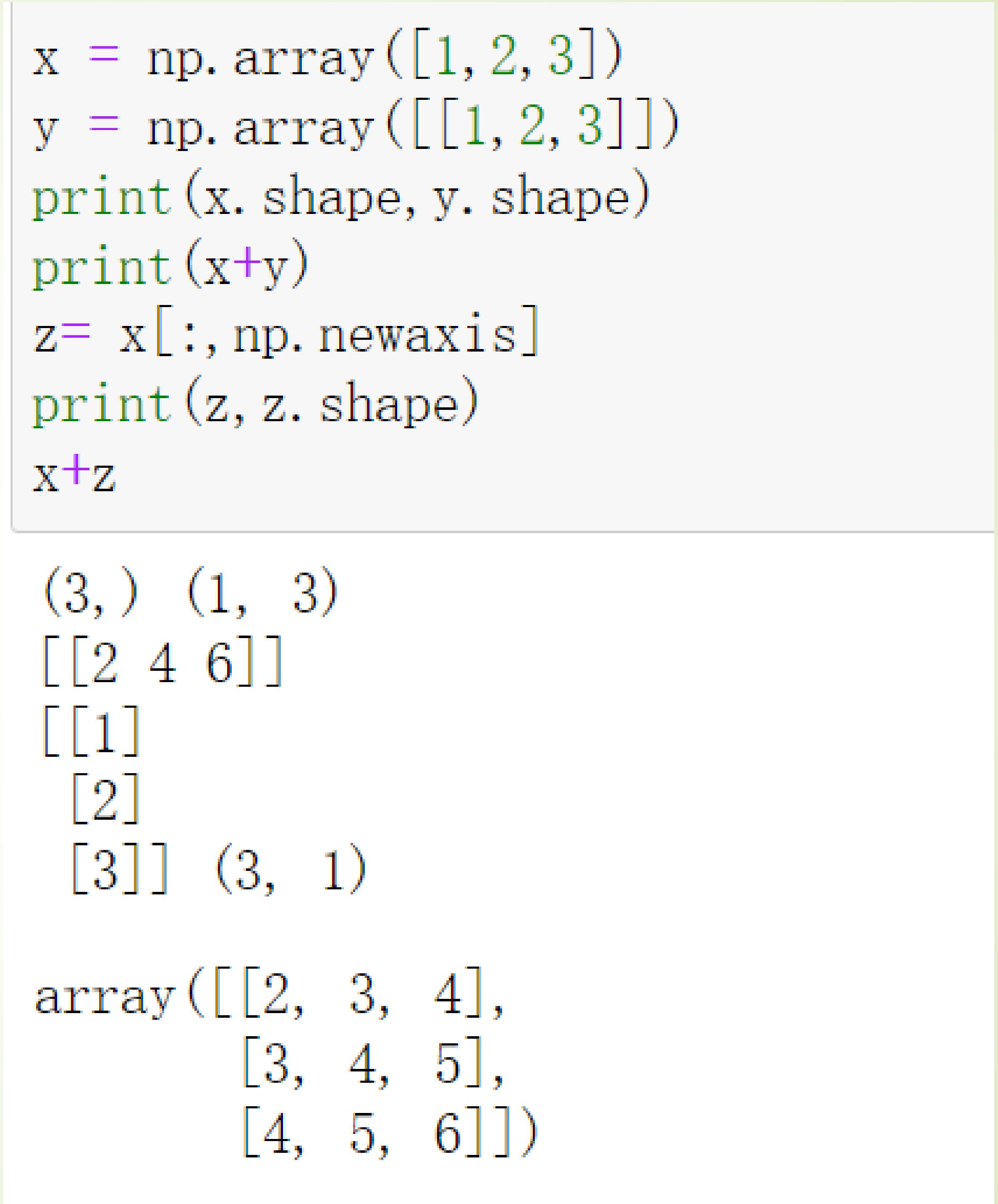
Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs (where an element is generally a scalar, but can be a vector or higher-order sub-array for generalized ufuncs). Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on. Broadcasting can be understood by four rules:

```
a = np.array([[1, 2, 3, 4], [10, 20, 30, 40]])
b = np.array([100, 100, 100, 100])
print(a + b)        # 行 对矩阵广播

c = np.array([3])
a * c               # 元素 对矩阵广播
```

```
[[101 102 103 104]
 [110 120 130 140]]

array([[  3,   6,   9,  12],
       [ 30,  60,  90, 120]])
```



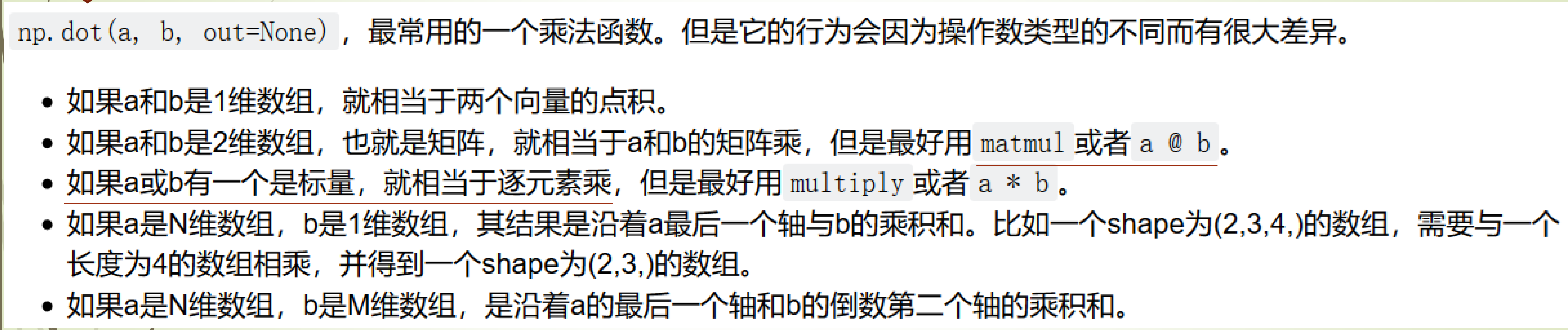
行-列交叉广播：



```
x = np.array([1,2,3])
y = np.array([[1,2,3]])
print(x.shape,y.shape)
print(x+y)
z= x[:,np.newaxis]
print(z,z.shape)
x+z
```

```
(3,) (1, 3)
[[2 4 6]]
[[1]
 [2]
 [3]] (3, 1)

array([[2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])
```

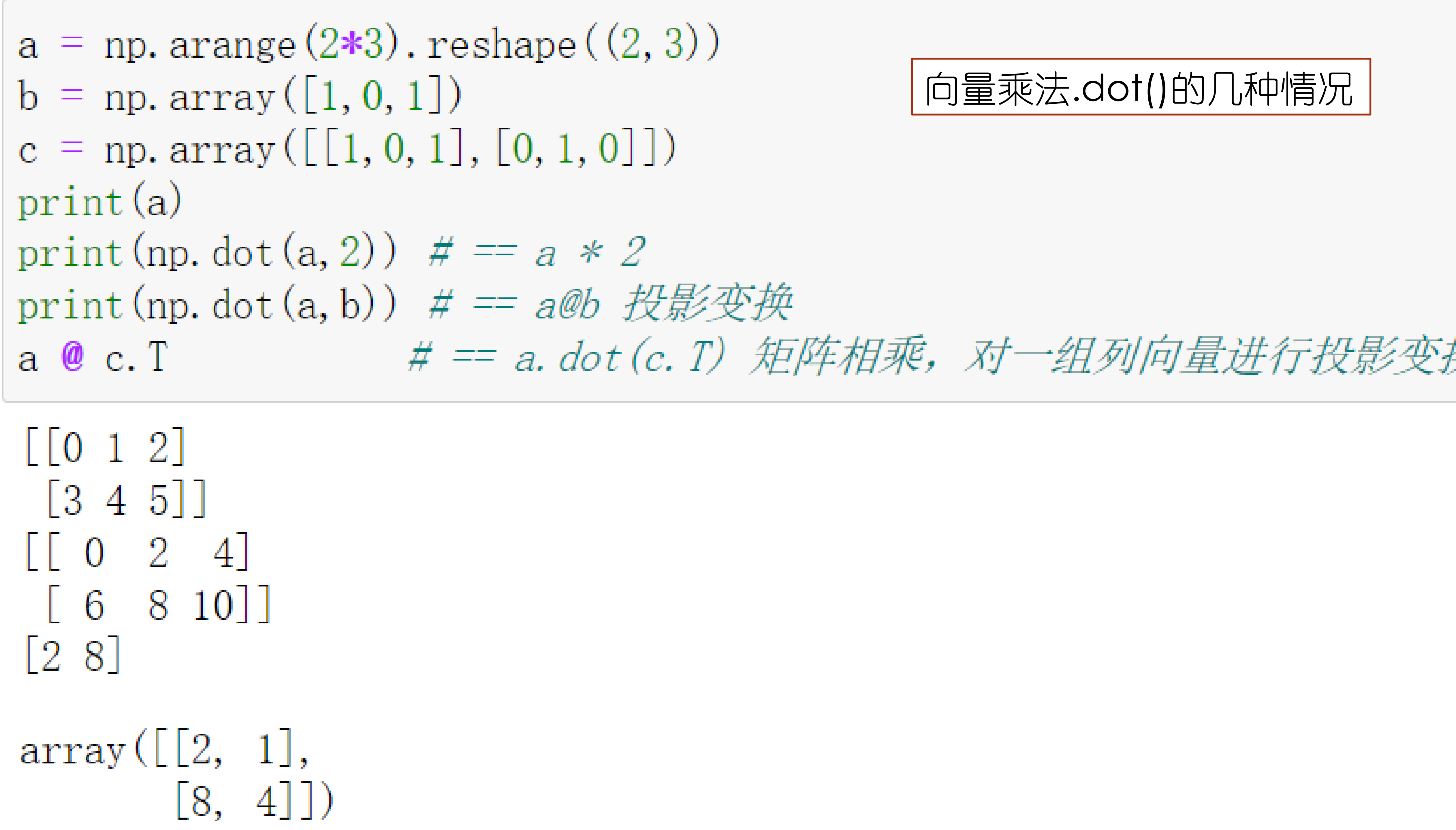
矩阵和向量乘法



`np.dot(a, b, out=None)`，最常用的一个乘法函数。但是它的行为会因为操作数类型的不同而有很大差异。

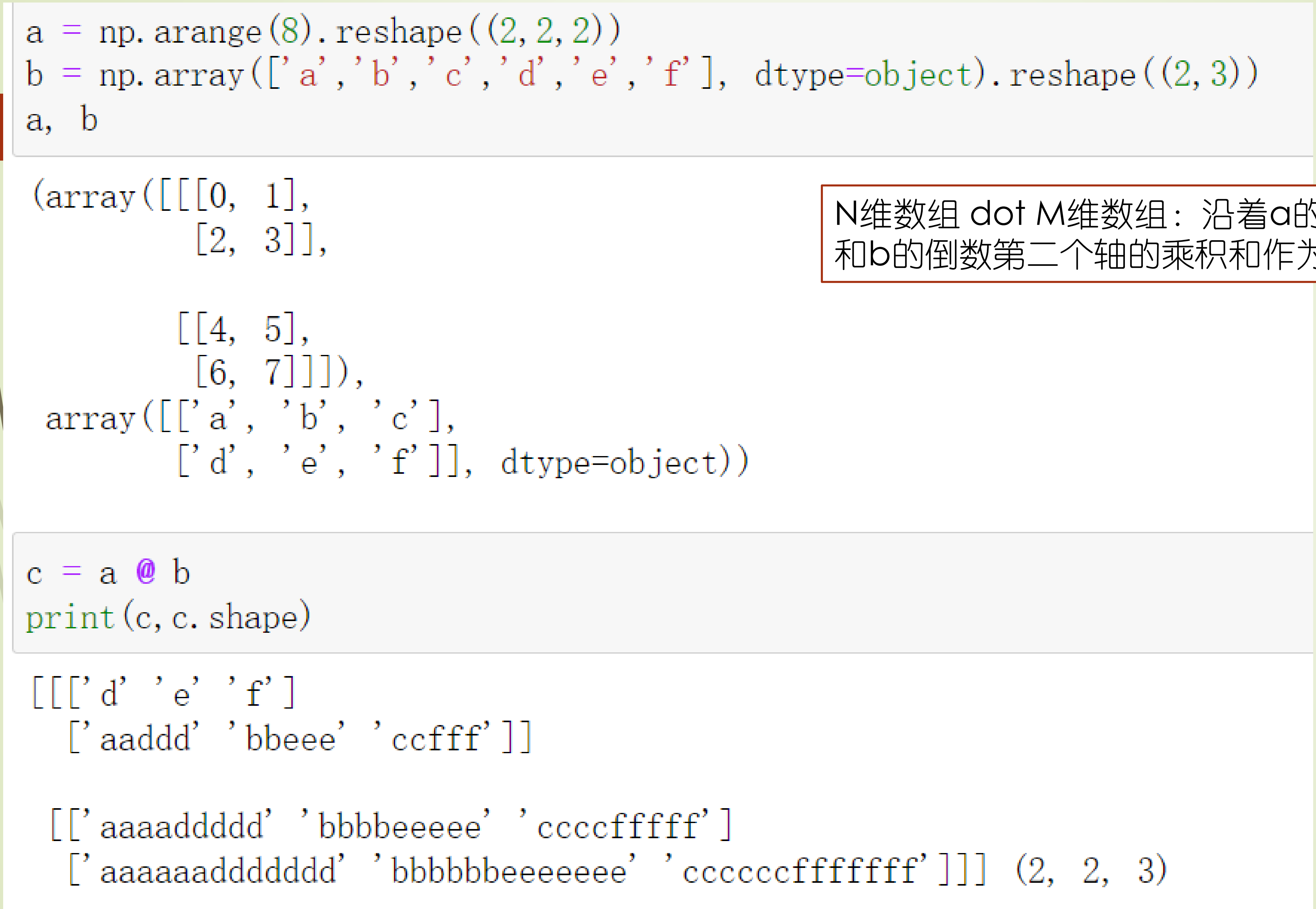
- 如果a和b是1维数组，就相当于两个向量的点积。
- 如果a和b是2维数组，也就是矩阵，就相当于a和b的矩阵乘，但是最好用 `matmul` 或者 `a @ b`。
- 如果a或b有一个是标量，就相当于逐元素乘，但是最好用 `multiply` 或者 `a * b`。
- 如果a是N维数组，b是1维数组，其结果是沿着a最后一个轴与b的乘积和。比如一个shape为(2,3,4,)的数组，需要与一个长度为4的数组相乘，并得到一个shape为(2,3,)的数组。
- 如果a是N维数组，b是M维数组，是沿着a的最后一个轴和b的倒数第二个轴的乘积和。

向量乘法.dot()的几种情况



```
a = np.arange(2*3).reshape((2,3))
b = np.array([1,0,1])
c = np.array([[1,0,1],[0,1,0]])
print(a)
print(np.dot(a,2)) # == a * 2
print(np.dot(a,b)) # == a@b 投影变换
a @ c.T           # == a.dot(c.T) 矩阵相乘，对一组列向量进行投影变
```

```
[[0 1 2]
 [3 4 5]]
[[ 0  2  4]
 [ 6  8 10]]
[2 8]

array([[2, 1],
       [8, 4]])
```



```python
a = np.arange(8).reshape((2,2,2))
b = np.array(['a','b','c','d','e','f'], dtype=object).reshape((2,3))
a, b
```

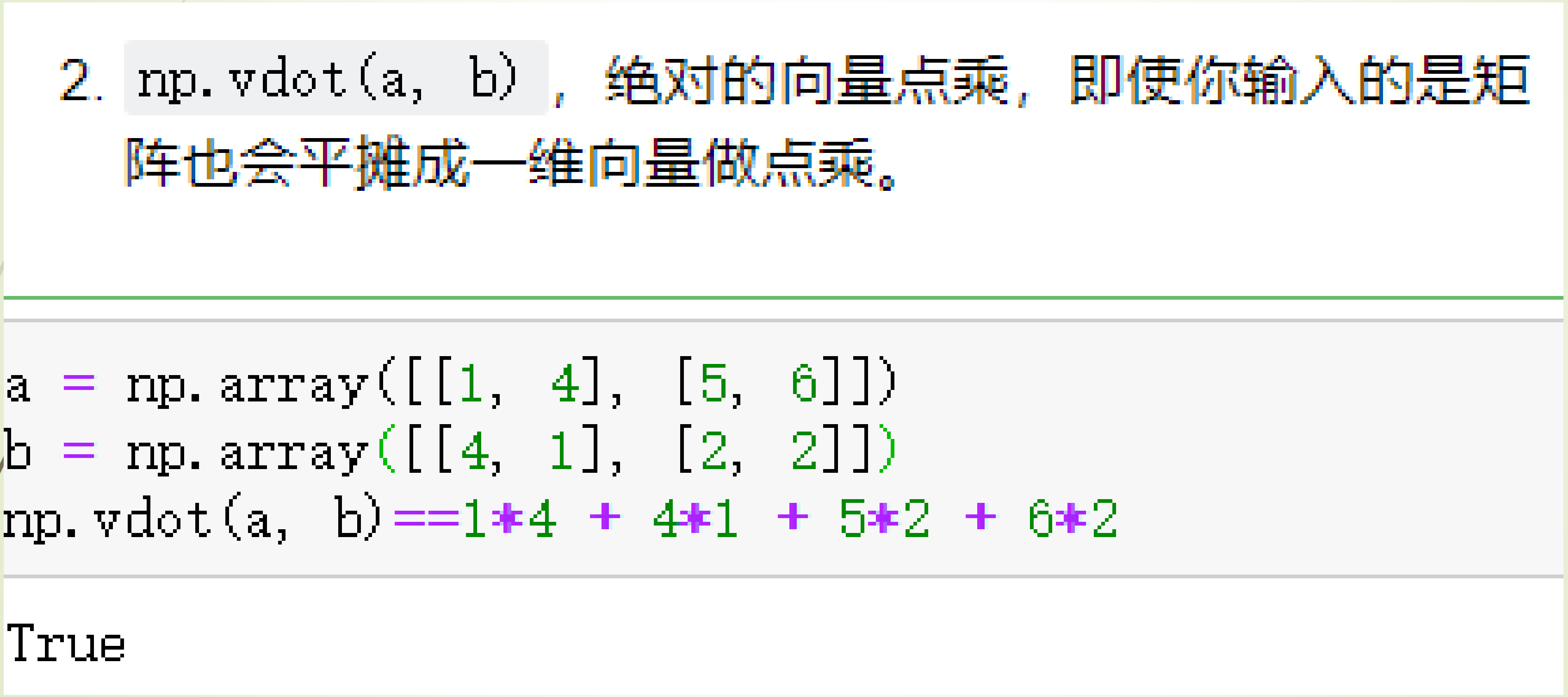
```
(array([[[0, 1],
         [2, 3]],

        [[4, 5],
         [6, 7]]]),
 array([['a', 'b', 'c'],
        ['d', 'e', 'f']], dtype=object))
```

```python
c = a @ b
print(c, c.shape)
```

```
[[['d' 'e' 'f']
  ['aaddd' 'bbeee' 'ccfff']]

 [['aaaadddd' 'bbbbeeeee' 'ccccfffff']
  ['aaaaaaddddddd' 'bbbbbbeeeeeee' 'ccccccffffff']]] (2, 2, 3)
```

N维数组 dot M维数组：沿着a的最后一个轴和b的倒数第二个轴的乘积和作为元素的广播

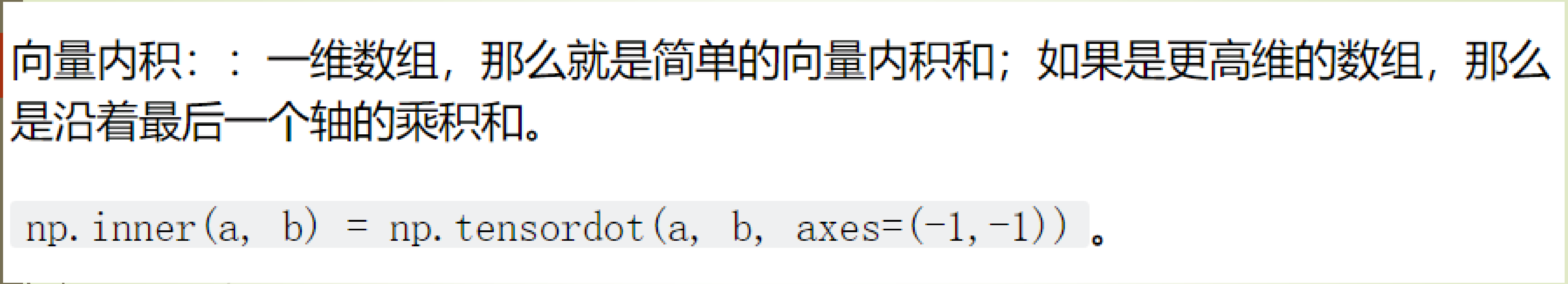
计算高维矩阵的逐元素投影量



2. `np.vdot(a, b)`，绝对的向量点乘，即使你输入的是矩阵也会平摊成一维向量做点乘。

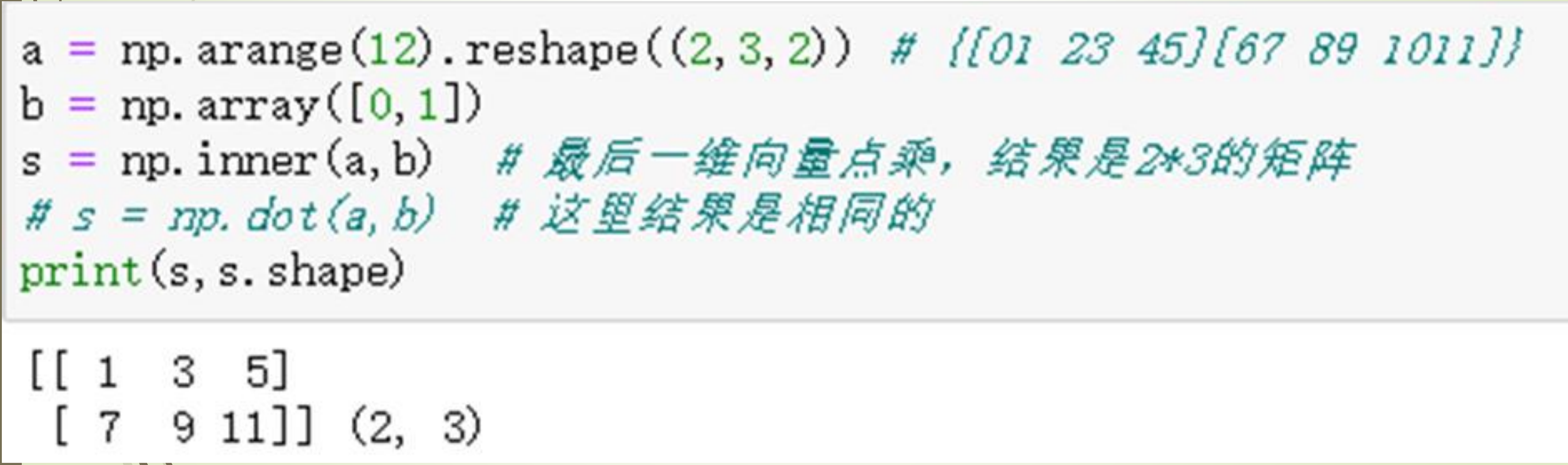
```
a = np.array([[1, 4], [5, 6]])
b = np.array([[4, 1], [2, 2]])
np.vdot(a, b)==1*4 + 4*1 + 5*2 + 6*2
```

```
True
```



向量内积：：一维数组，那么就是简单的向量内积和；如果是更高维的数组，那么是沿着最后一个轴的乘积和。

`np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))`。



```
a = np.arange(12).reshape((2,3,2)) # [[01  23  45][67  89  1011]]
b = np.array([0,1])
s = np.inner(a,b)  # 最后一维向量点乘，结果是2*3的矩阵
# s = np.dot(a,b)  # 这里结果是相同的
print(s,s.shape)
```

```
[[ 1  3  5]
 [ 7  9 11]] (2, 3)
```

```python
a = np.arange(8).reshape((2,2,2)) # [01 23][45 67]
b = np.array(['a','b','c','d'], dtype=object).reshape((2,2))
```

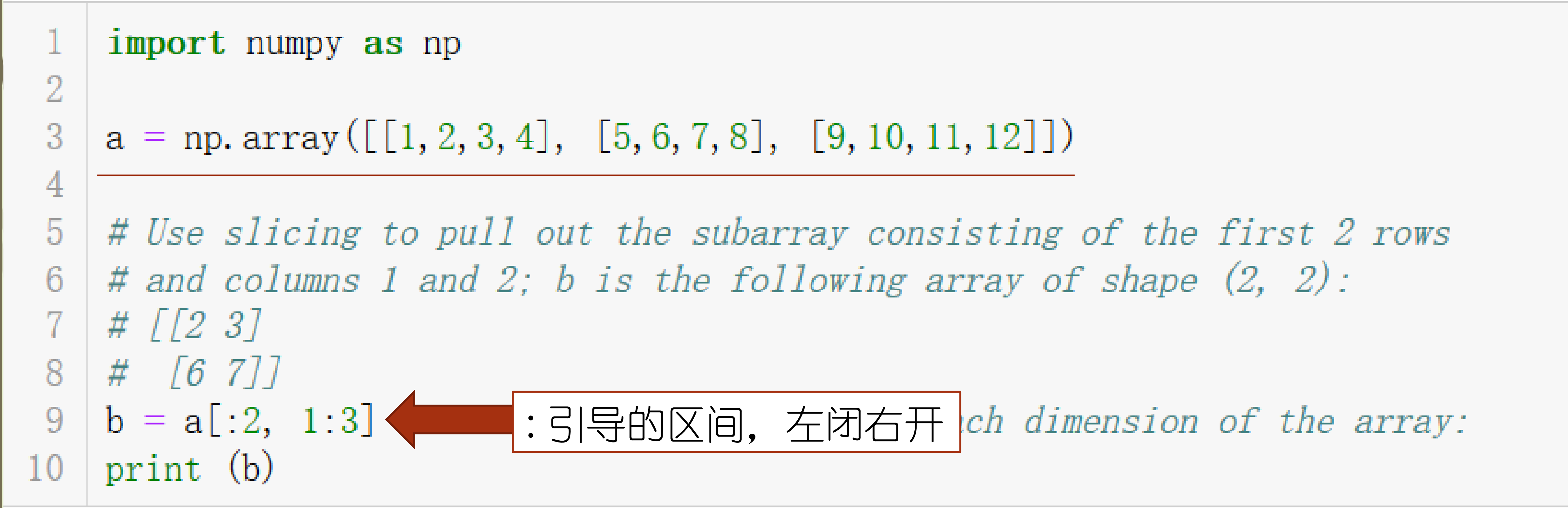
```python
np.inner(a, b) # 对应字符串相乘相加
```

```
array([[['b', 'd'],
        ['aabbb', 'ccddd']],

       [['aaaabbbbb', 'ccccddddd'],
        ['aaaaaabbbbbbb', 'ccccccddddddd']]], dtype=object)
```

```python
a.dot(b)  # 这里就不同了（注意，和矩阵乘法的规则是sigma i*j inner product 的规则是 sigma
```

```
array([[['c', 'd'],
        ['aaccc', 'bbddd']],

       [['aaaaccccc', 'bbbbddddd'],
        ['aaaaaaccccccc', 'bbbbbbddddddd']]], dtype=object)
```

矩阵的下标访问与切片 :Array indexing and Slicing

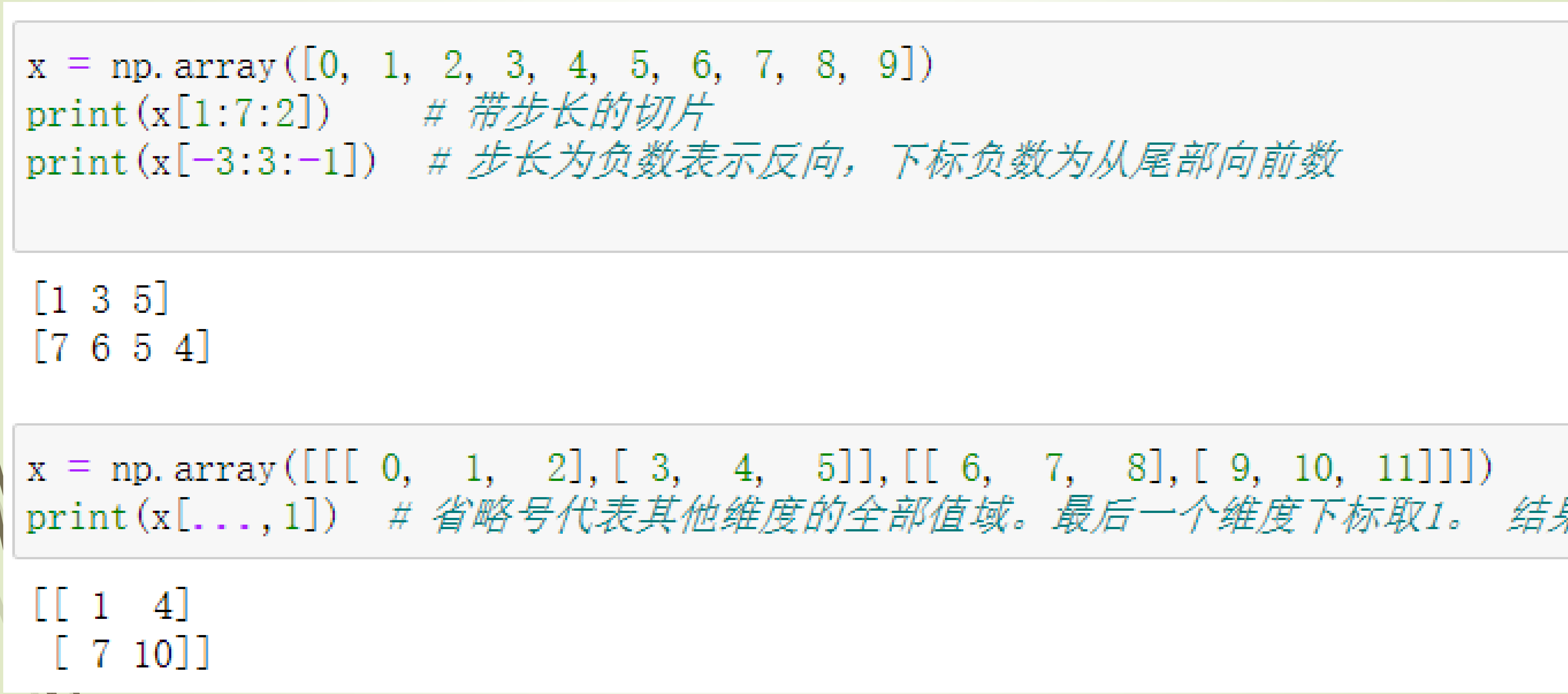


```python
import numpy as np

a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]                                    ch dimension of the array:
print (b)
```

: 引导的区间，左闭右开

```
[[2 3]
 [6 7]]
```

矩阵切片的步长与区间方向：



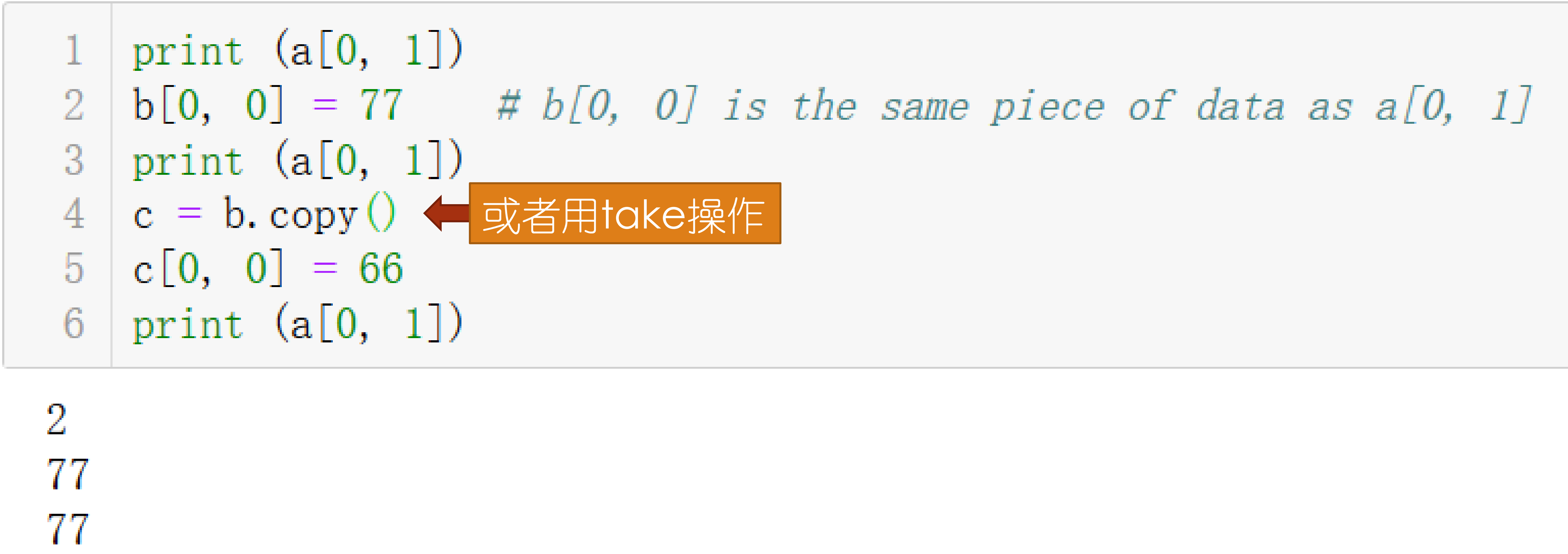
```
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x[1:7:2])     # 带步长的切片
print(x[-3:3:-1])   # 步长为负数表示反向，下标负数为从尾部向前数
```

```
[1 3 5]
[7 6 5 4]
```

```
x = np.array([[[ 0,   1,   2],[ 3,   4,   5]],[[ 6,   7,   8],[ 9,  10,  11]]])
print(x[...,1])  # 省略号代表其他维度的全部值域。最后一个维度下标取1。 结
```

```
[[ 1  4]
 [ 7 10]]
```

矩阵切片视图：A slice of an array is a view(视图) into the same data, modifying it will modify the original array

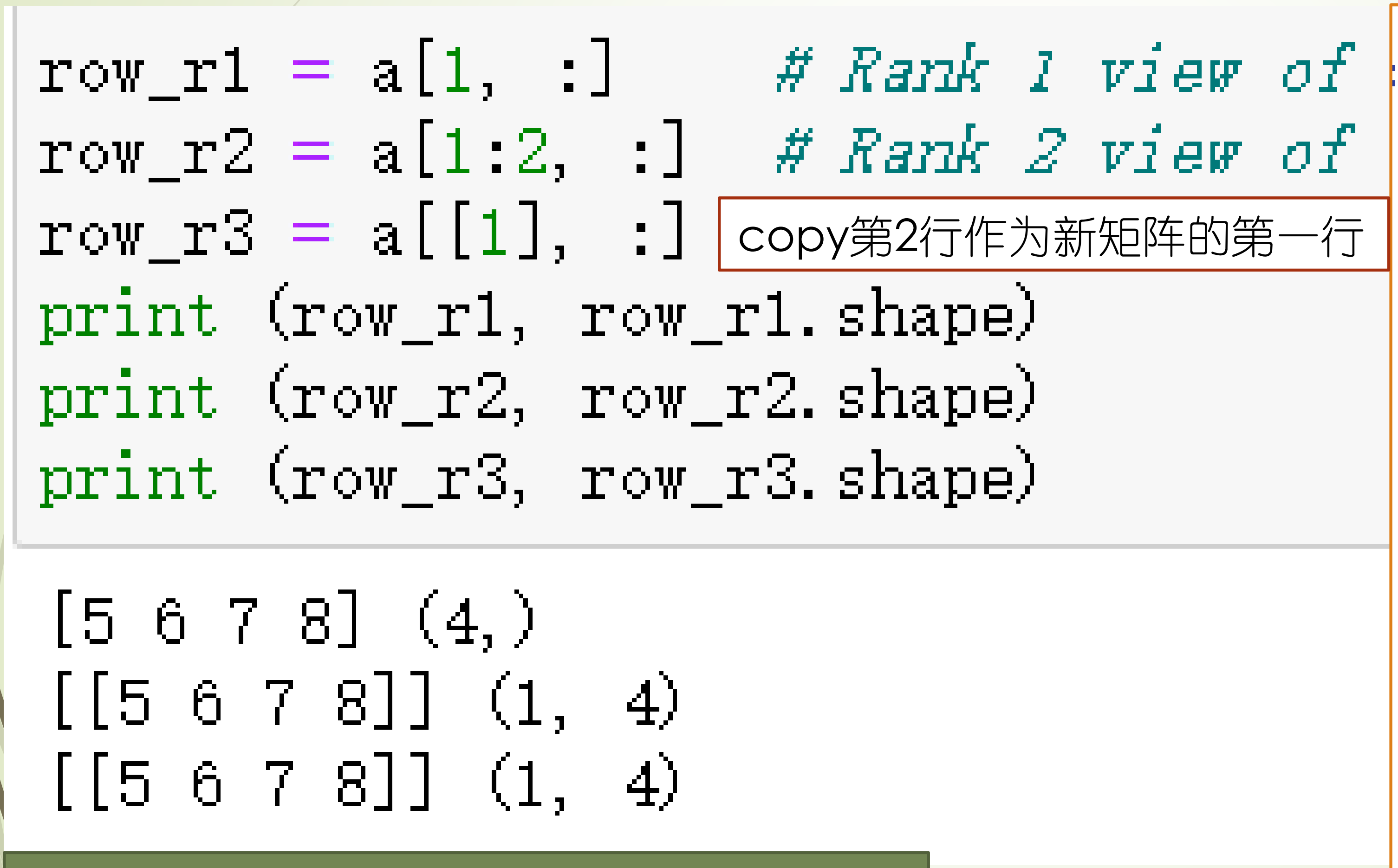


```python
print (a[0, 1])
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print (a[0, 1])
c = b.copy()
c[0, 0] = 66
print (a[0, 1])
```

或者用take操作

```
2
77
77
```

- 切片操作如果直接标定一个维度，会得到一个低一阶的矩阵
- 每个维度都是区段，哪怕区段里只有一个下标
- 如果切片参数为一个list，则默认采用高级索引模式，copy

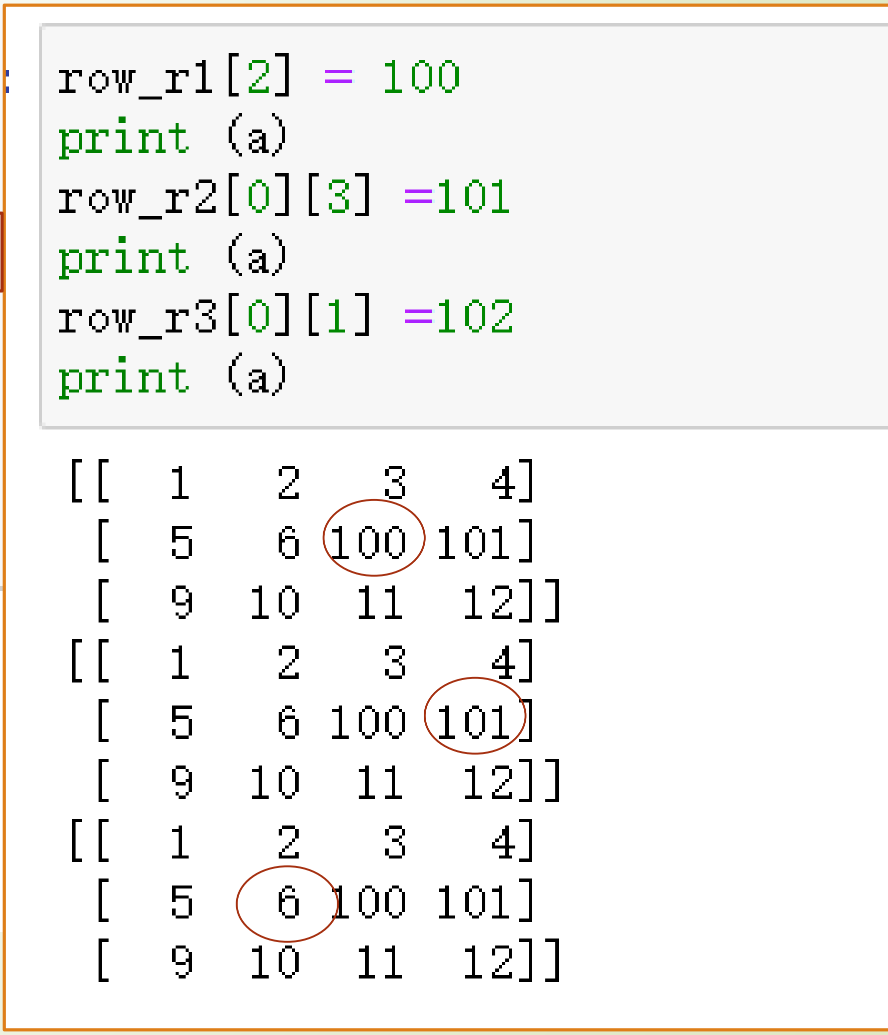


```
row_r1 = a[1, :]      # Rank 1 view of
row_r2 = a[1:2, :]    # Rank 2 view of
row_r3 = a[[1], :]
print (row_r1, row_r1.shape)
print (row_r2, row_r2.shape)
print (row_r3, row_r3.shape)
```

copy第2行作为新矩阵的第一行

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

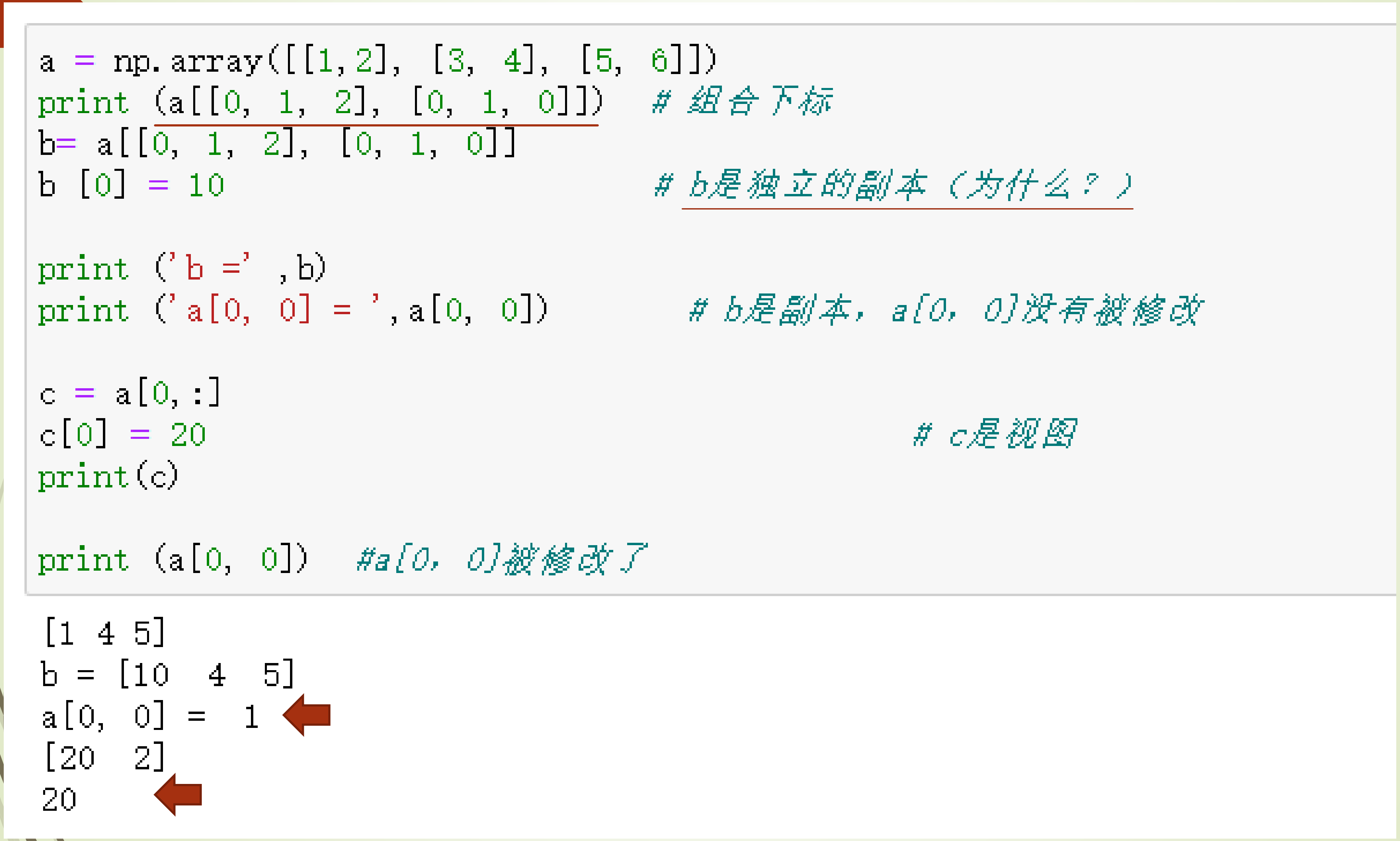
是否是独立的副本可以通过 .base属性
和 .flags.owndata属性来确认



```
row_r1[2] = 100
print (a)
row_r2[0][3] =101
print (a)
row_r3[0][1] =102
print (a)
```

```
[[  1   2   3   4]
 [  5   6 100 101]
 [  9  10  11  12]]
[[  1   2   3   4]
 [  5   6 100 101]
 [  9  10  11  12]]
[[  1   2   3   4]
 [  5   6 100 101]
 [  9  10  11  12]]
```

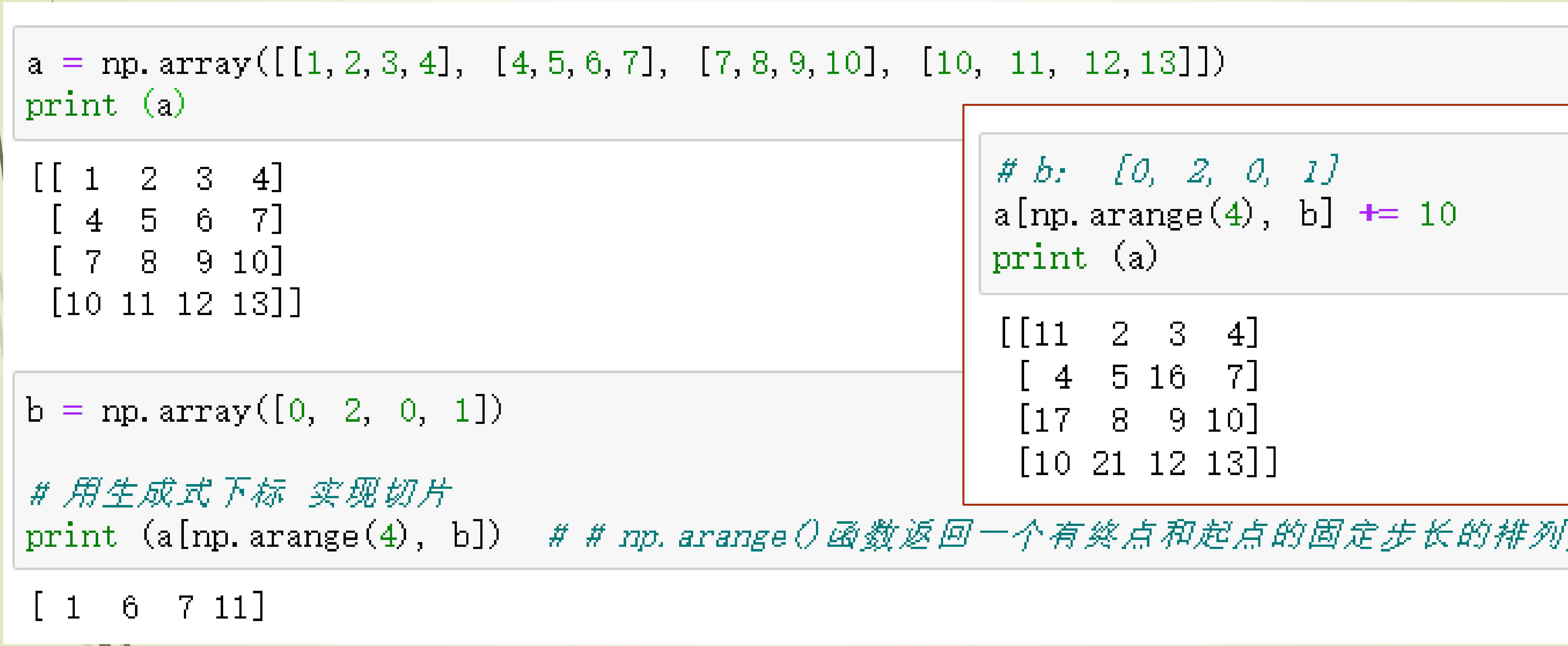
advance indexing：通过下标元组集合来标定元素



```
a = np.array([[1,2], [3, 4], [5, 6]])
print (a[[0, 1, 2], [0, 1, 0]])  # 组合下标
b= a[[0, 1, 2], [0, 1, 0]]
b [0] = 10                       # b是独立的副本（为什么？）

print ('b =' ,b)
print ('a[0, 0] = ',a[0, 0])       # b是副本，a[0, 0]没有被修改

c = a[0,:]
c[0] = 20                              # c是视图
print(c)

print (a[0, 0])  #a[0, 0]被修改了
```

```
[1 4 5]
b = [10  4  5]
a[0, 0] =  1
[20  2]
20
```

生成式也可以用来做组合下标

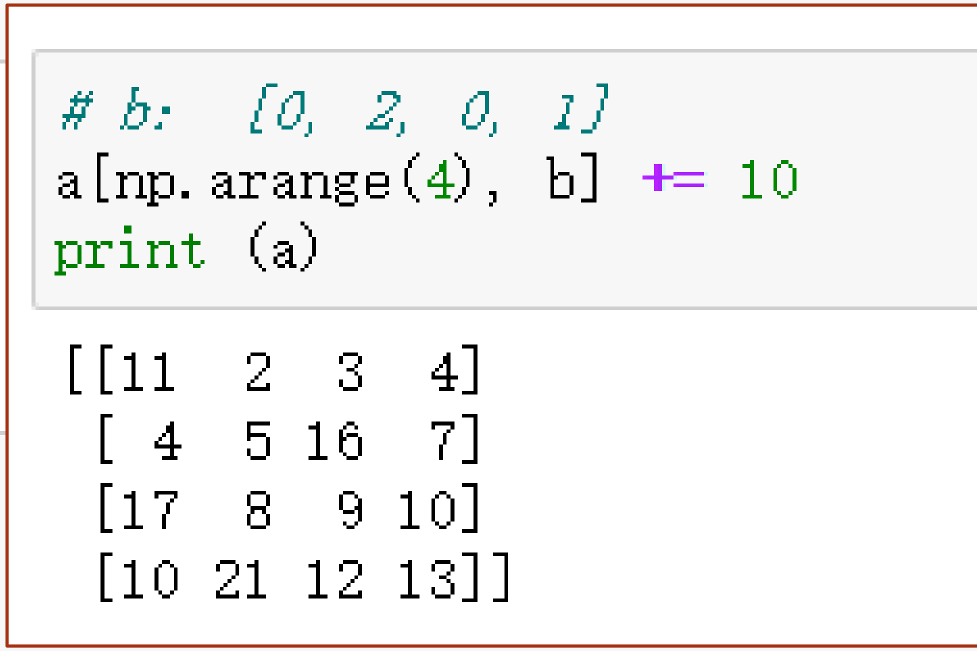


```
a = np.array([[1,2,3,4], [4,5,6,7], [7,8,9,10], [10, 11, 12,13]])
print (a)
```

```
[[ 1  2  3  4]
 [ 4  5  6  7]
 [ 7  8  9 10]
 [10 11 12 13]]
```

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

# 用生成式下标 实现切片
print (a[np.arange(4), b])   # # np.arange()函数返回一个有终点和起点的固定步长的排列
```

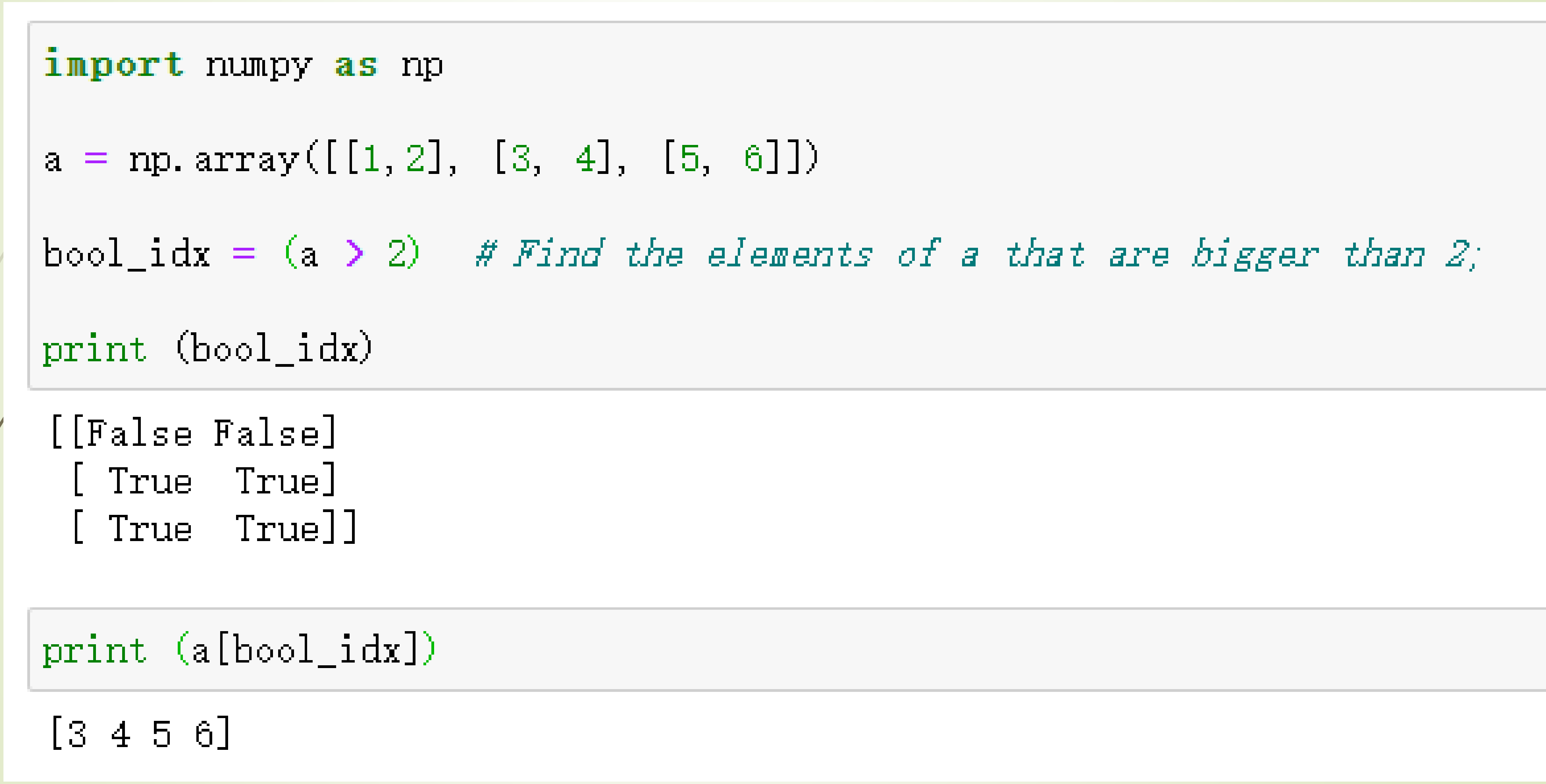
```
[ 1  6  7 11]
```



```
# b:  [0, 2, 0, 1]
a[np.arange(4), b] += 10
print (a)
```

```
[[11  2  3  4]
 [ 4  5 16  7]
 [17  8  9 10]
 [10 21 12 13]]
```

通过条件约束获得布尔下标矩阵

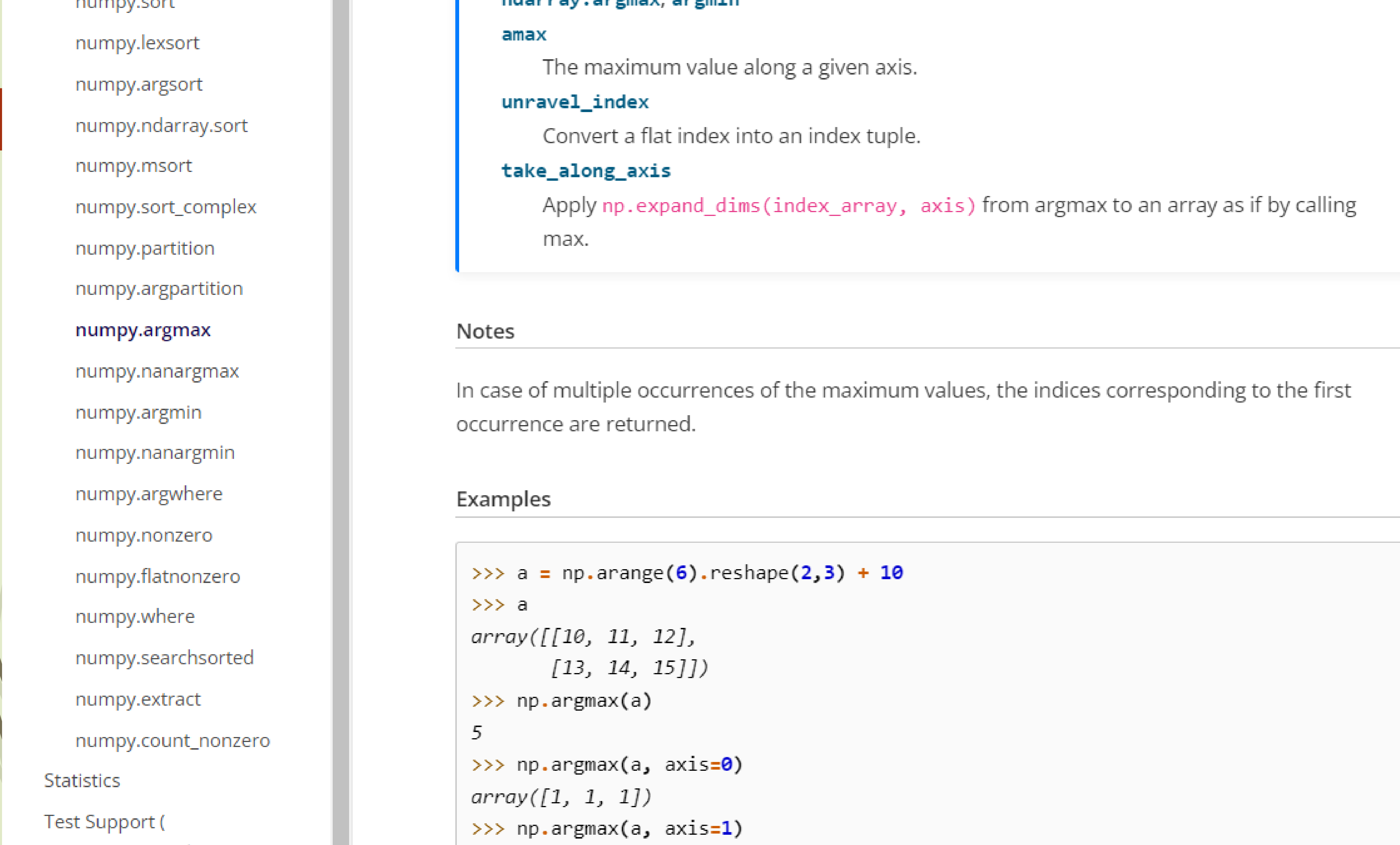


```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;

print (bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

```
print (a[bool_idx])
```

```
[3 4 5 6]
```



numpy.sort

numpy.lexsort

numpy.argsort

numpy.ndarray.sort

numpy.msort

numpy.sort_complex

numpy.partition

numpy.argpartition

numpy.argmax

numpy.nanargmax

numpy.argmin

numpy.nanargmin

numpy.argwhere

numpy.nonzero

numpy.flatnonzero

numpy.where

numpy.searchsorted

numpy.extract

numpy.count_nonzero

Statistics

Test Support (

`ndarray.argmax`, `argmin`

`amax`
The maximum value along a given axis.

`unravel_index`
Convert a flat index into an index tuple.

`take_along_axis`
Apply `np.expand_dims(index_array, axis)` from argmax to an array as if by calling max.

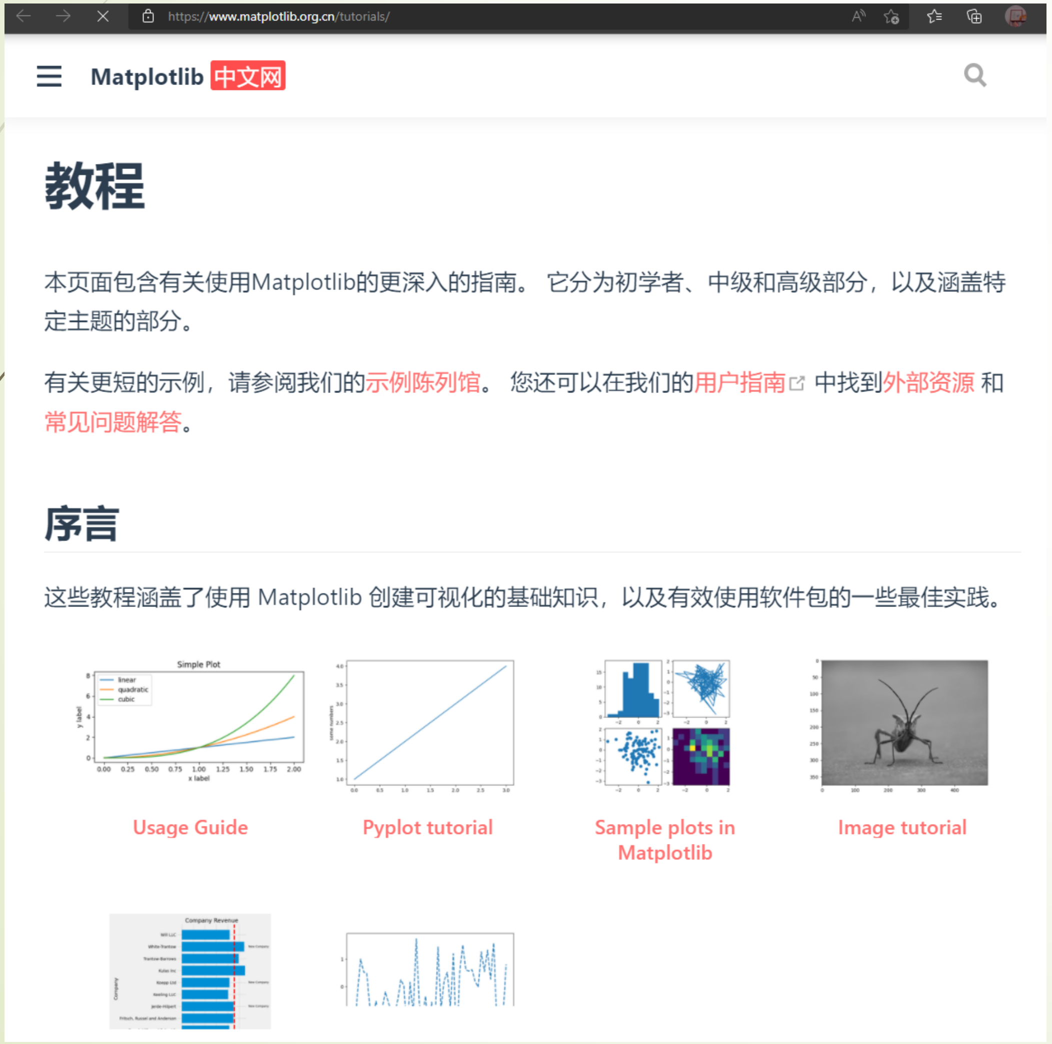
Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
```

可视化软件包matplotlib https://www.matplotlib.org.cn/tutorials/


https://www.matplotlib.org.cn/tutorials/

Matplotlib 中文网

教程

本页面包含有关使用Matplotlib的更深入的指南。 它分为初学者、中级和高级部分，以及涵盖特定主题的部分。

有关更短的示例，请参阅我们的示例陈列馆。 您还可以在我们的用户指南 中找到外部资源 和常见问题解答。

序言

这些教程涵盖了使用 Matplotlib 创建可视化的基础知识，以及有效使用软件包的一些最佳实践。

Importing Matplotlib

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

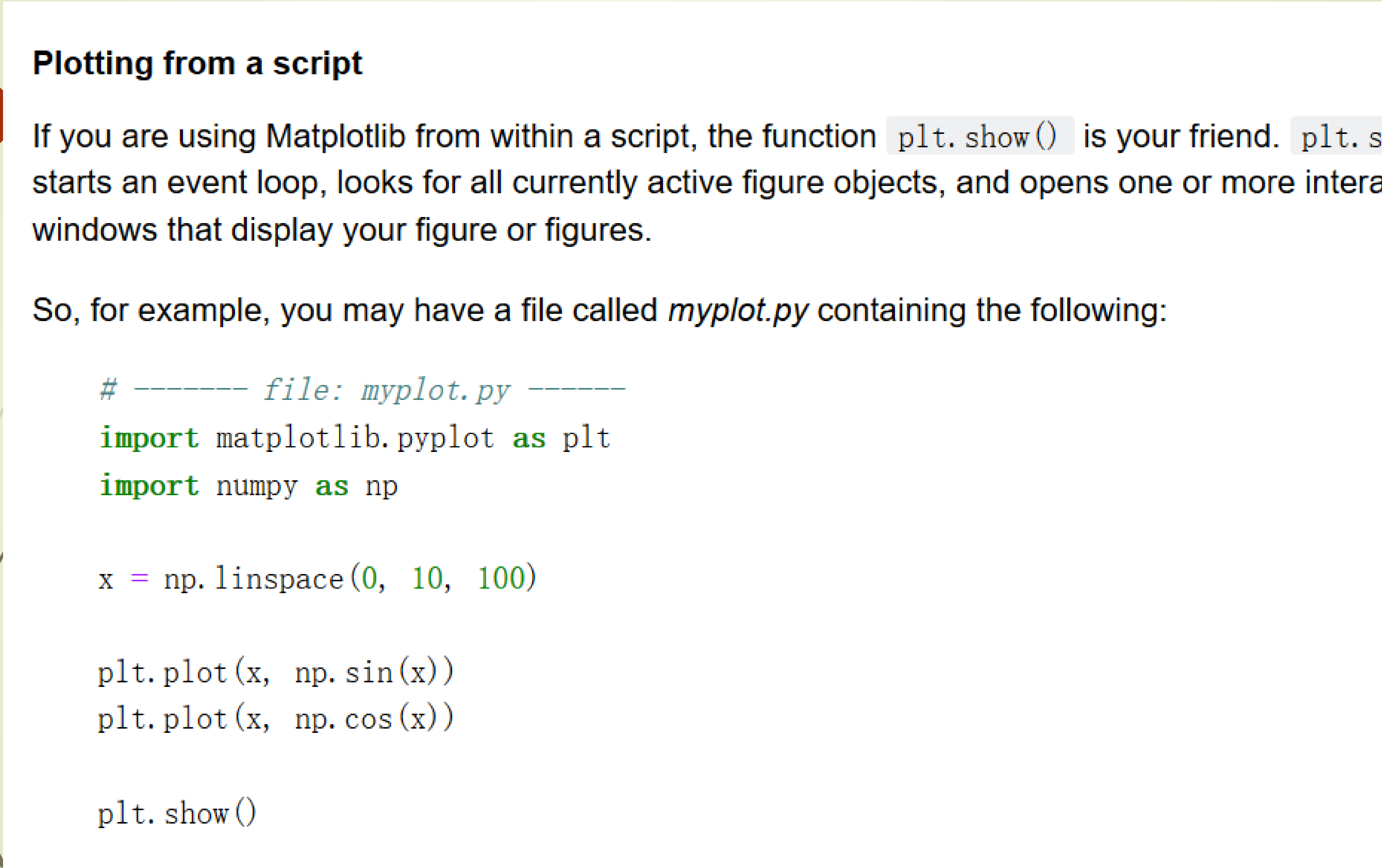
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

The `plt` interface is what we will use most often, as we shall see throughout this chapter.

Setting Styles

We will use the `plt.style` directive to choose appropriate aesthetic styles for our figures. Here we will set the `classic` style, which ensures that the plots we create use the classic Matplotlib style:

```
plt.style.use('classic')
```



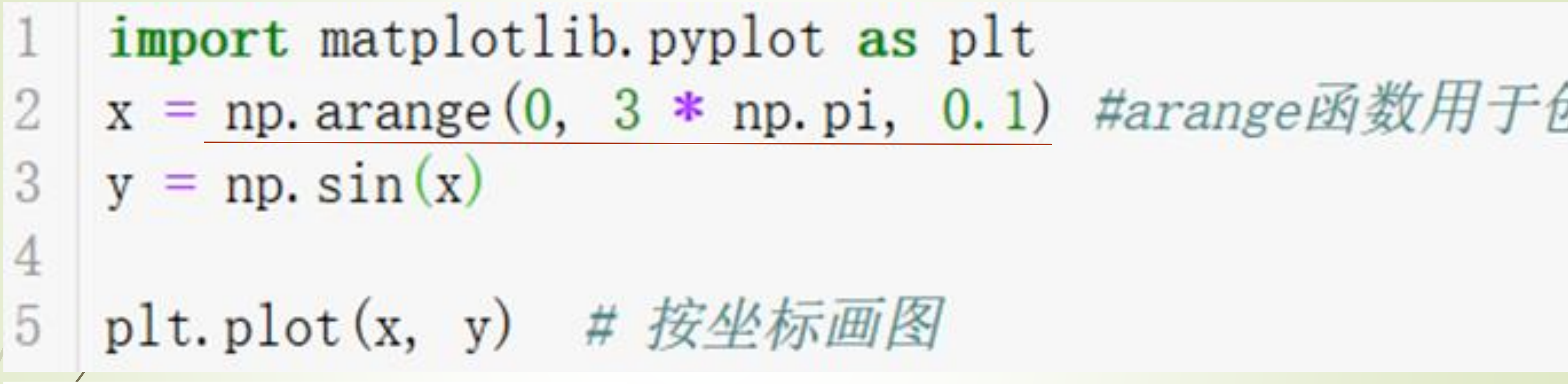
Plotting from a script

If you are using Matplotlib from within a script, the function `plt.show()` is your friend. `plt.s`
starts an event loop, looks for all currently active figure objects, and opens one or more intera
windows that display your figure or figures.

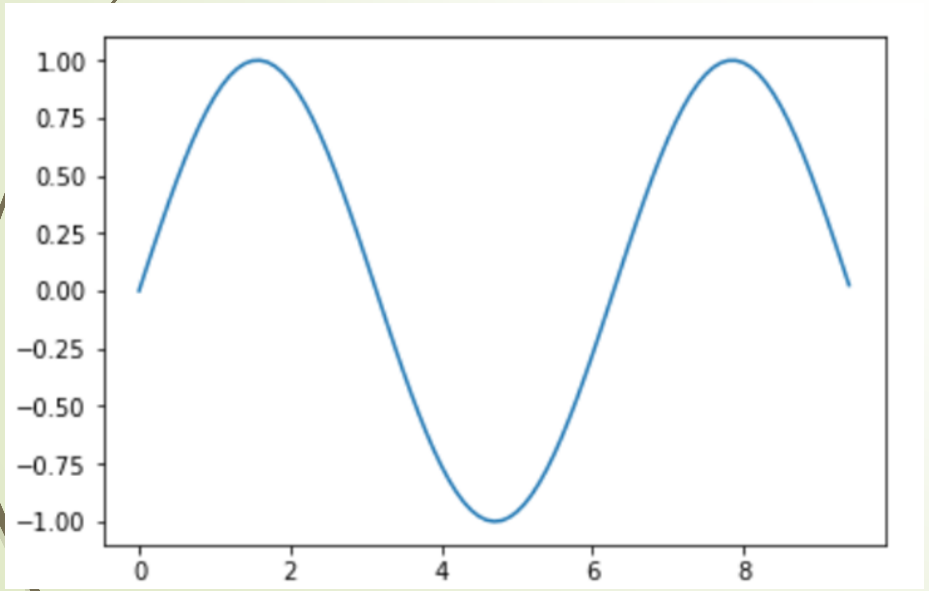
So, for example, you may have a file called *myplot.py* containing the following:

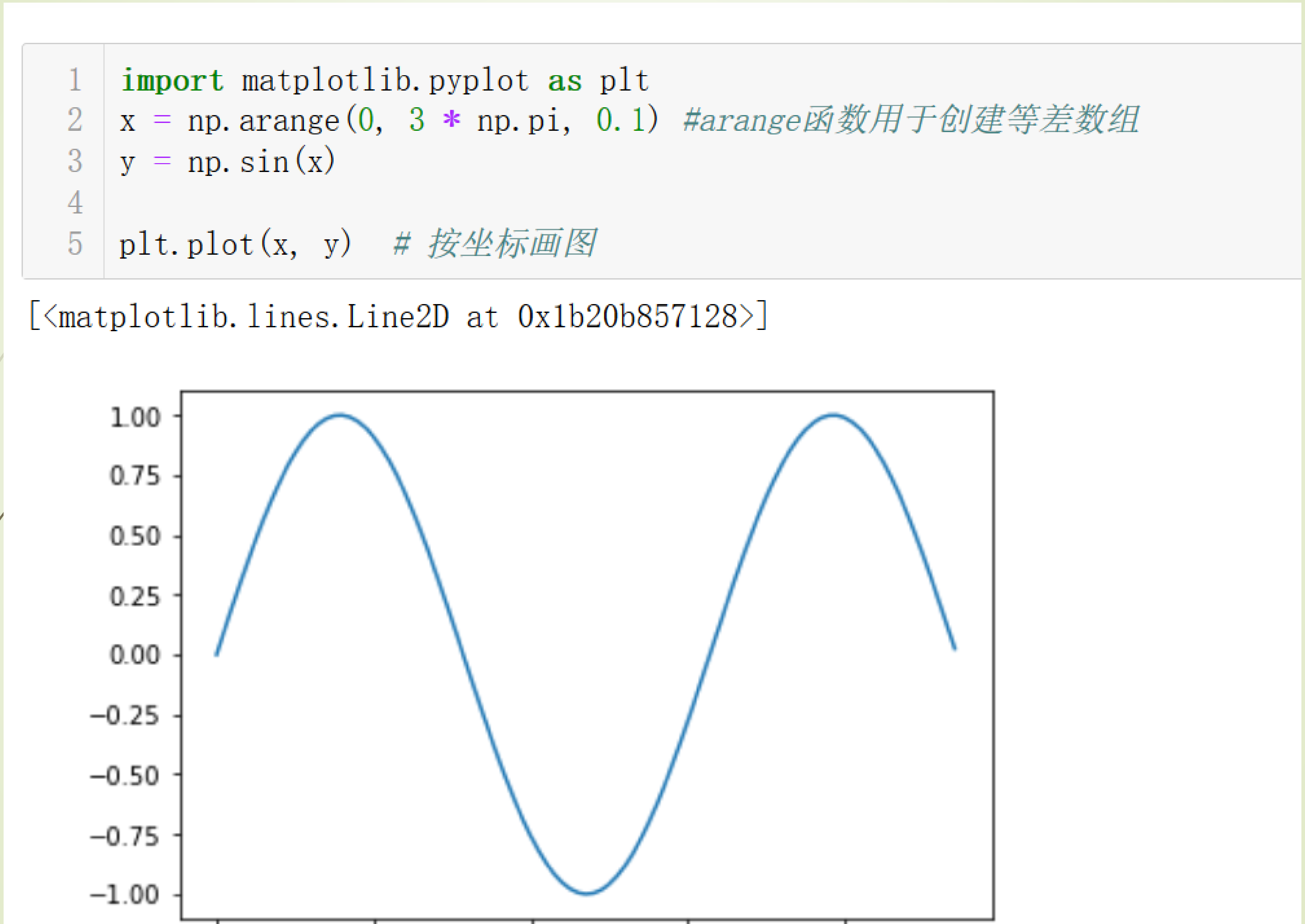
```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

Plotting —— 二维画图软件包



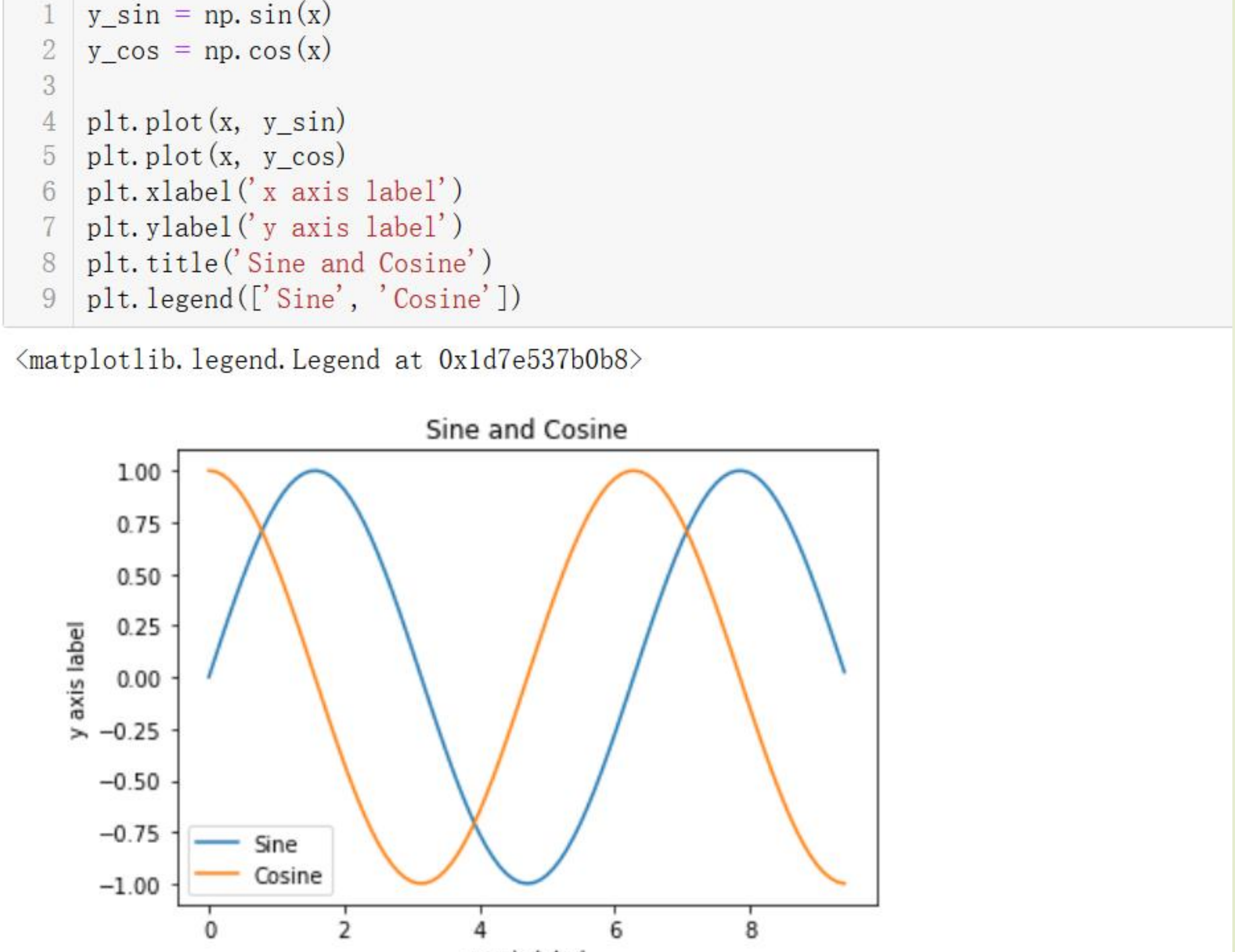
```
import matplotlib.pyplot as plt
x = np.arange(0, 3 * np.pi, 0.1) #arange函数用于
y = np.sin(x)

plt.plot(x, y)   # 按坐标画图
```





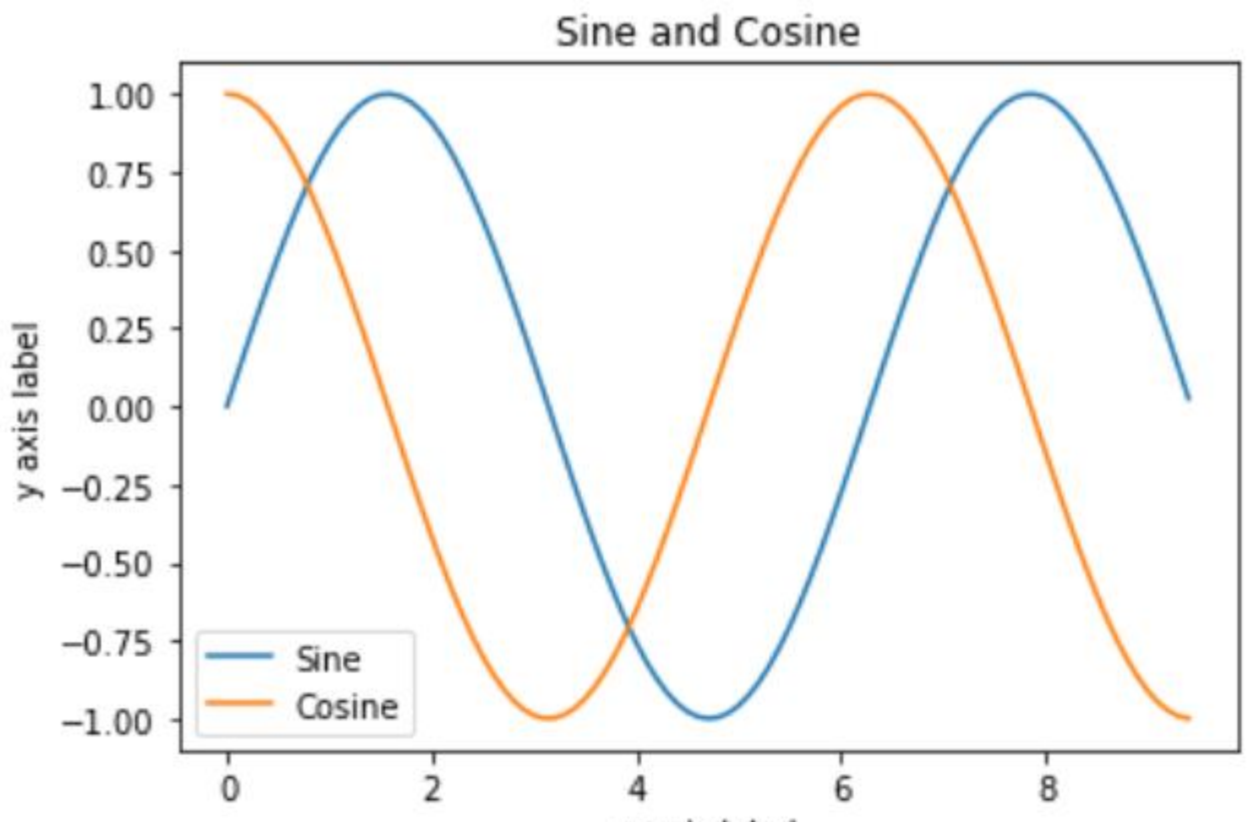
```
import matplotlib.pyplot as plt
x = np.arange(0, 3 * np.pi, 0.1) #arange函数用于创建等差数组
y = np.sin(x)

plt.plot(x, y)  # 按坐标画图
```

```
[<matplotlib.lines.Line2D at 0x1b20b857128>]
```

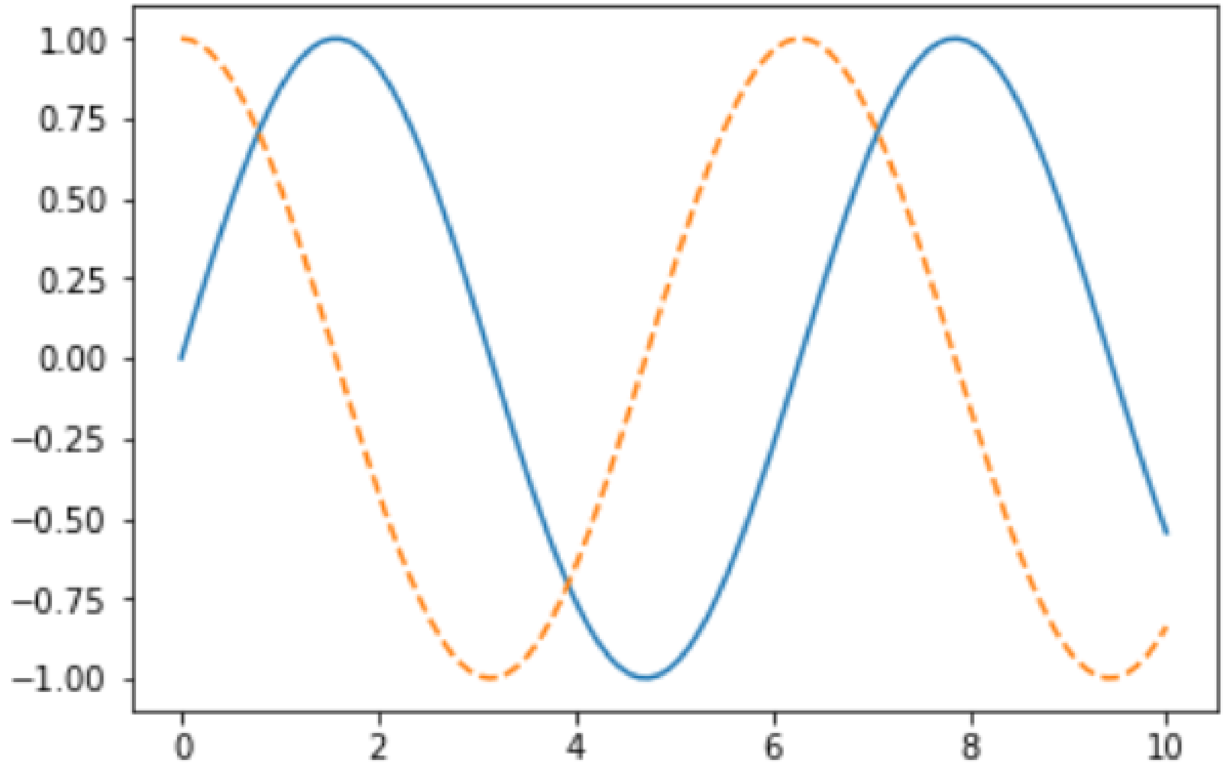


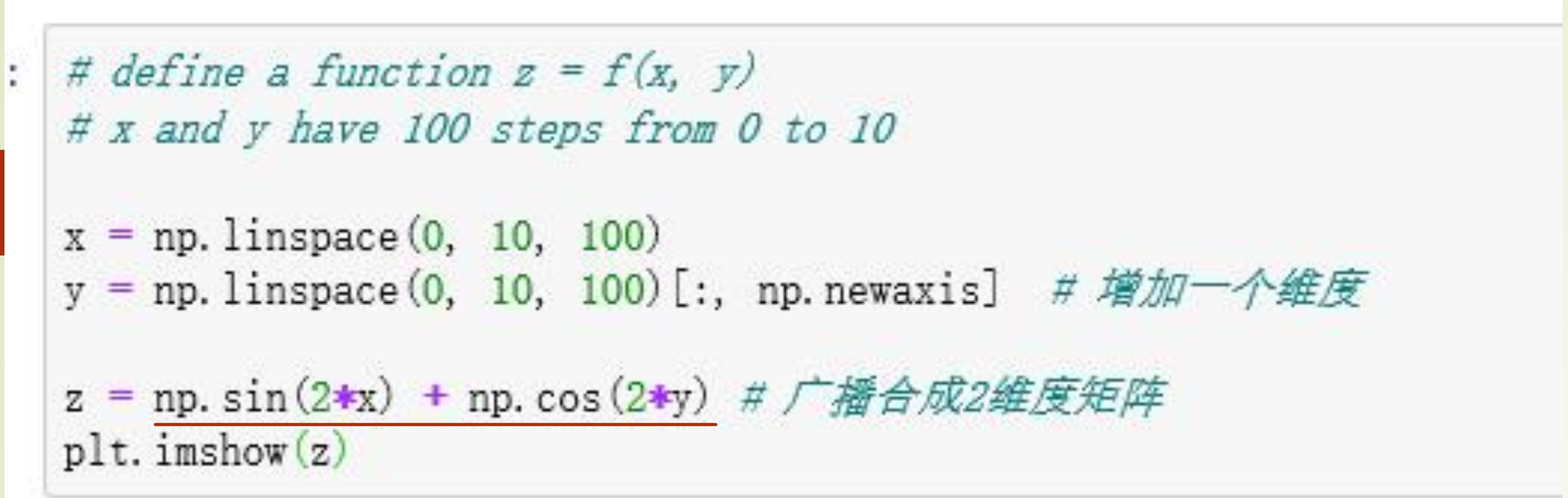
```python
y_sin = np.sin(x)
y_cos = np.cos(x)

plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

```
<matplotlib.legend.Legend at 0x1d7e537b0b8>
```



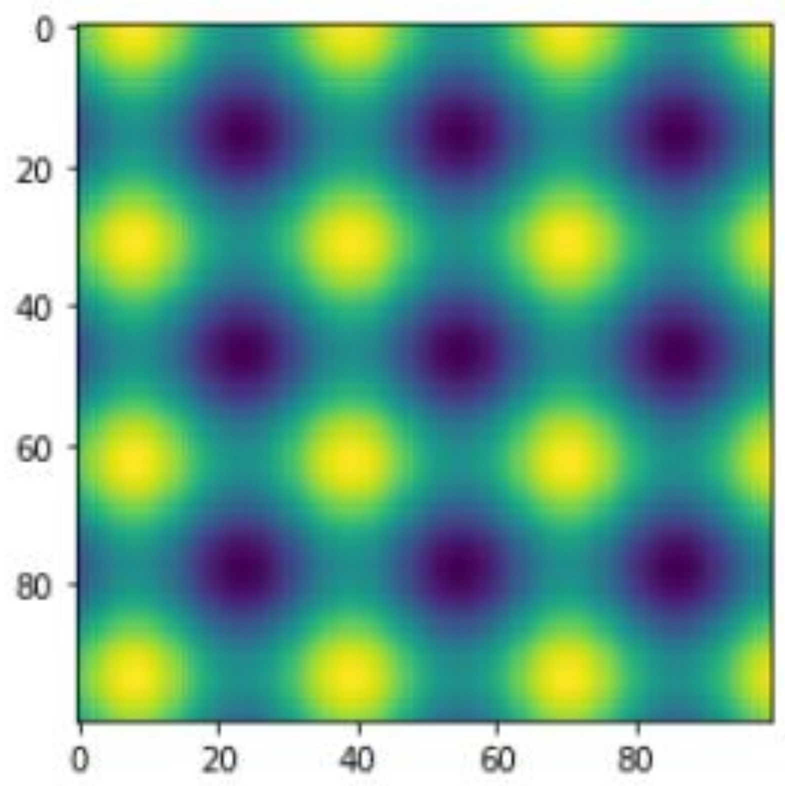
```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 100)

#fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```

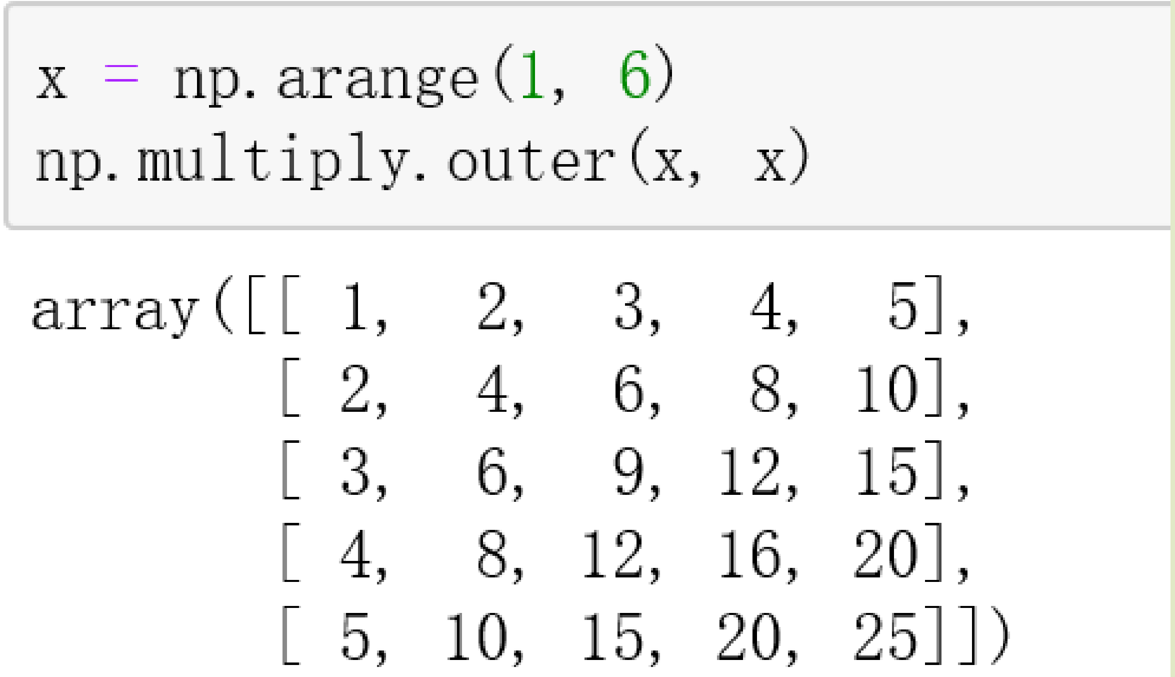




```
# define a function z = f(x, y)
# x and y have 100 steps from 0 to 10

x = np.linspace(0, 10, 100)
y = np.linspace(0, 10, 100)[:, np.newaxis]  # 增加一个维度

z = np.sin(2*x) + np.cos(2*y)  # 广播合成2维度矩阵
plt.imshow(z)
```

```
<matplotlib.image.AxesImage at 0x23bca086c88>
```





```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the `savefig()` command. For example, to save the previous figure as a PNG file, you can run this:

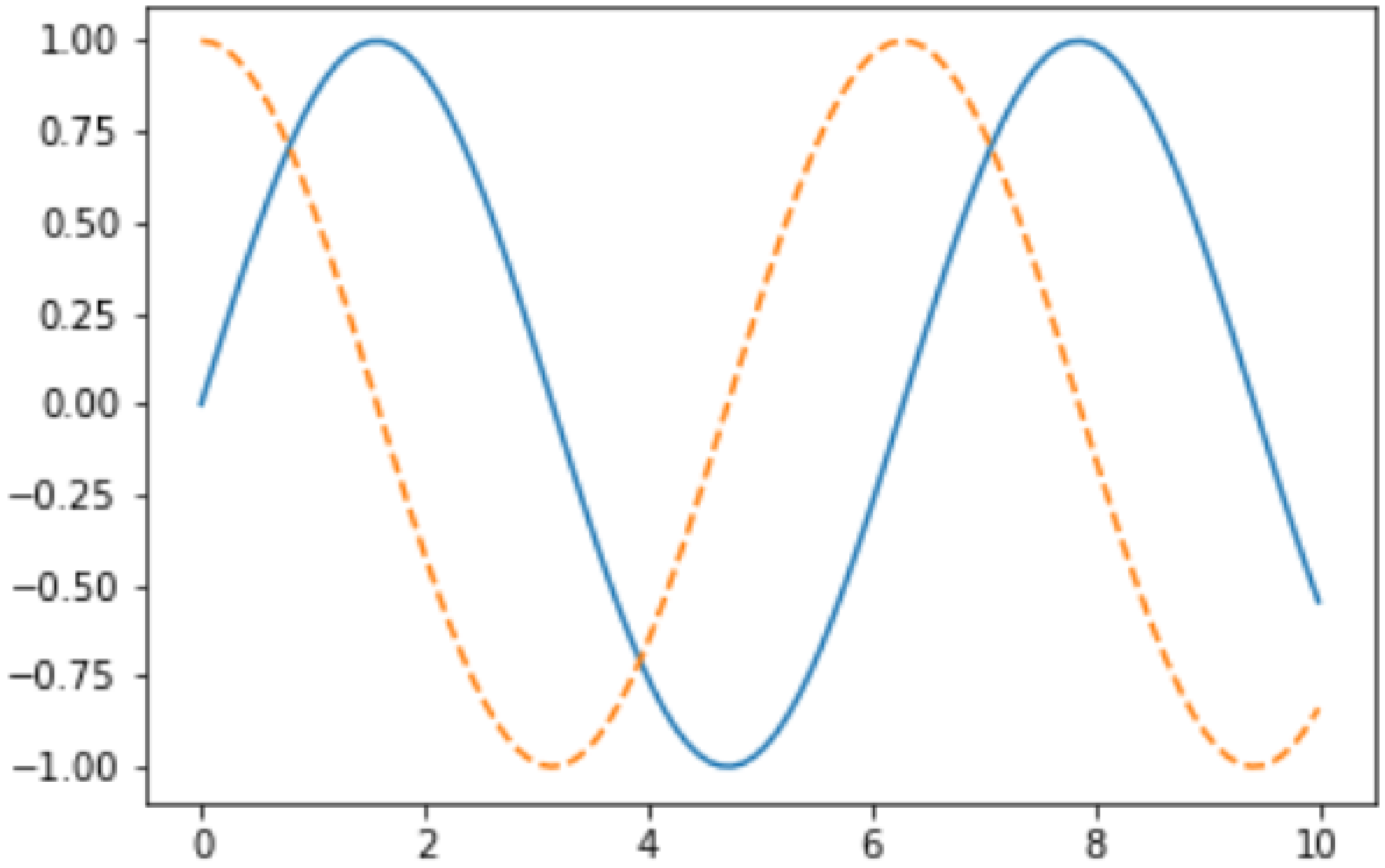
```
fig.savefig('my_figure.png')
```

We now have a file called `my_figure.png` in the current working directory:

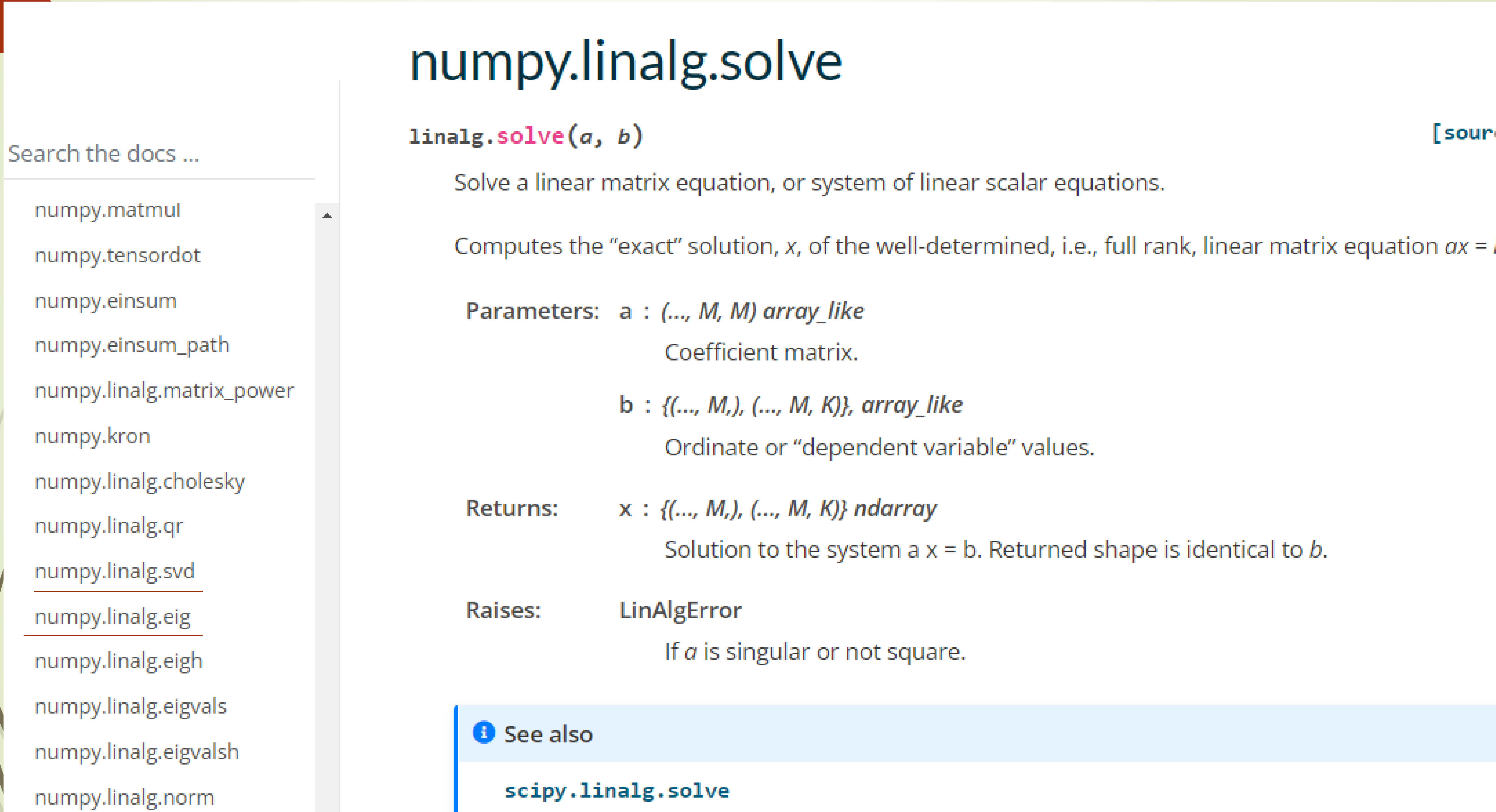
```
!dir  my_figure.png
```

```
 驱动器 C 中的卷是 Windows
 卷的序列号是 368A-7FED

 C:\Users\hjf_p\2019python\notebooks 的目录

2019/03/27  07:09            22,604 my_figure.png
               1 个文件         22,604 字节
               0 个目录 138,808,619,008 可用字节
```

```python
from IPython.display import Image
Image('my_figure.png')
```



numpy的线性代数包提供了矩阵计算功能



Search the docs ...

- numpy.matmul
- numpy.tensordot
- numpy.einsum
- numpy.einsum_path
- numpy.linalg.matrix_power
- numpy.kron
- numpy.linalg.cholesky
- numpy.linalg.qr
- numpy.linalg.svd
- numpy.linalg.eig
- numpy.linalg.eigh
- numpy.linalg.eigvals
- numpy.linalg.eigvalsh
- numpy.linalg.norm

numpy.linalg.solve

`linalg.solve(a, b)` [sour

Solve a linear matrix equation, or system of linear scalar equations.

Computes the "exact" solution, *x*, of the well-determined, i.e., full rank, linear matrix equation *ax* =

Parameters: **a** : *(..., M, M) array_like*
Coefficient matrix.

b : *{(..., M,), (..., M, K)}, array_like*
Ordinate or "dependent variable" values.

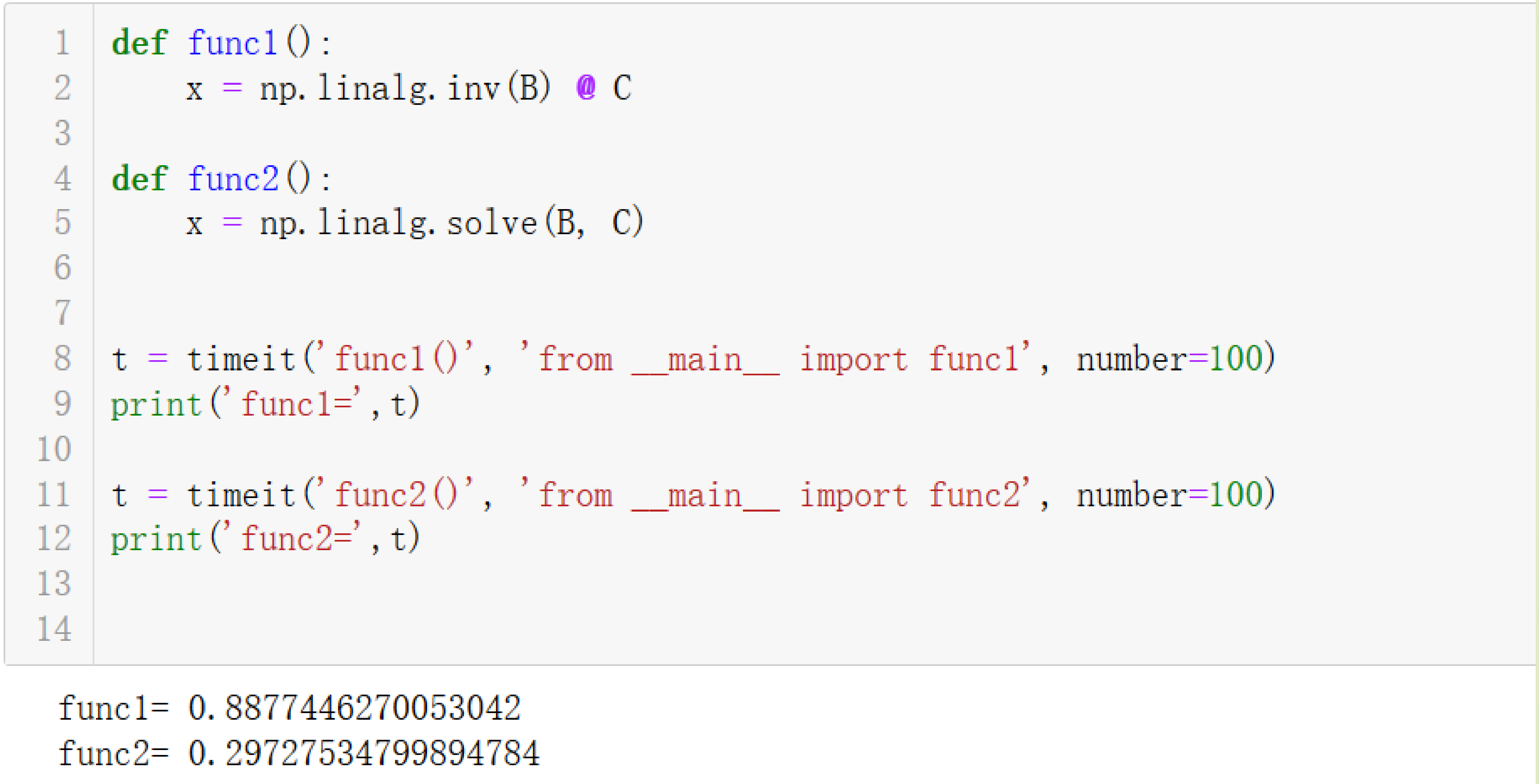
Returns: **x** : *{(..., M,), (..., M, K)} ndarray*
Solution to the system a x = b. Returned shape is identical to *b*.

Raises: LinAlgError
If *a* is singular or not square.

See also

`scipy.linalg.solve`

2 计算$Bx = C$的解。



```python
def func1():
    x = np.linalg.inv(B) @ C

def func2():
    x = np.linalg.solve(B, C)


t = timeit('func1()', 'from __main__ import func1', number=100)
print('func1=',t)

t = timeit('func2()', 'from __main__ import func2', number=100)
print('func2=',t)


```

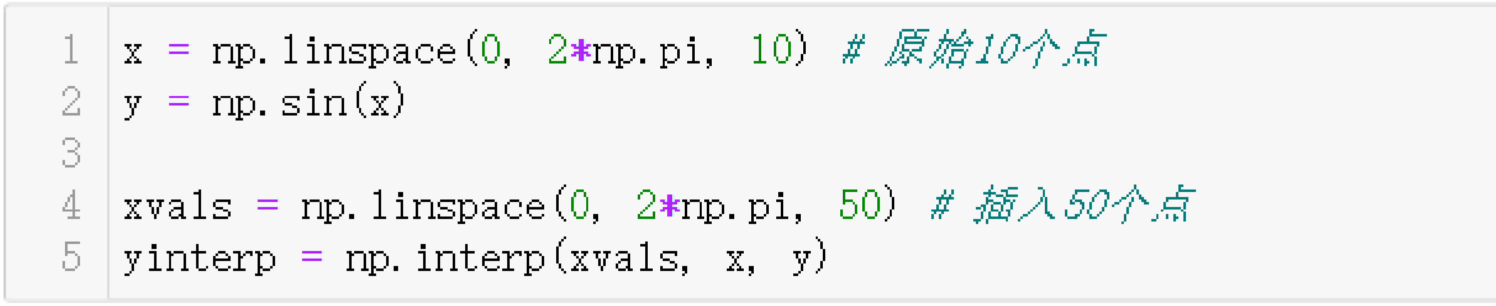
```
func1= 0.8877446270053042
func2= 0.29727534799894784
```

向量计算：

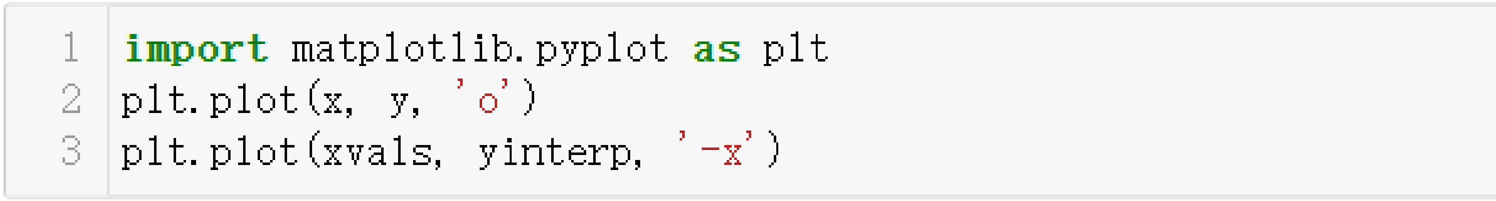
- 以向量计算为基础的计算模式相对于传统的循环遍历方案，更有利于发挥Numpy软件包的计算效率

函数插值：

- **numpy.interp(x, xp, fp, left=None, right=None, period=None)**
- 线性插值。xp默认是严格单调增的下标x， fp是对应的y。x是需要插值的点

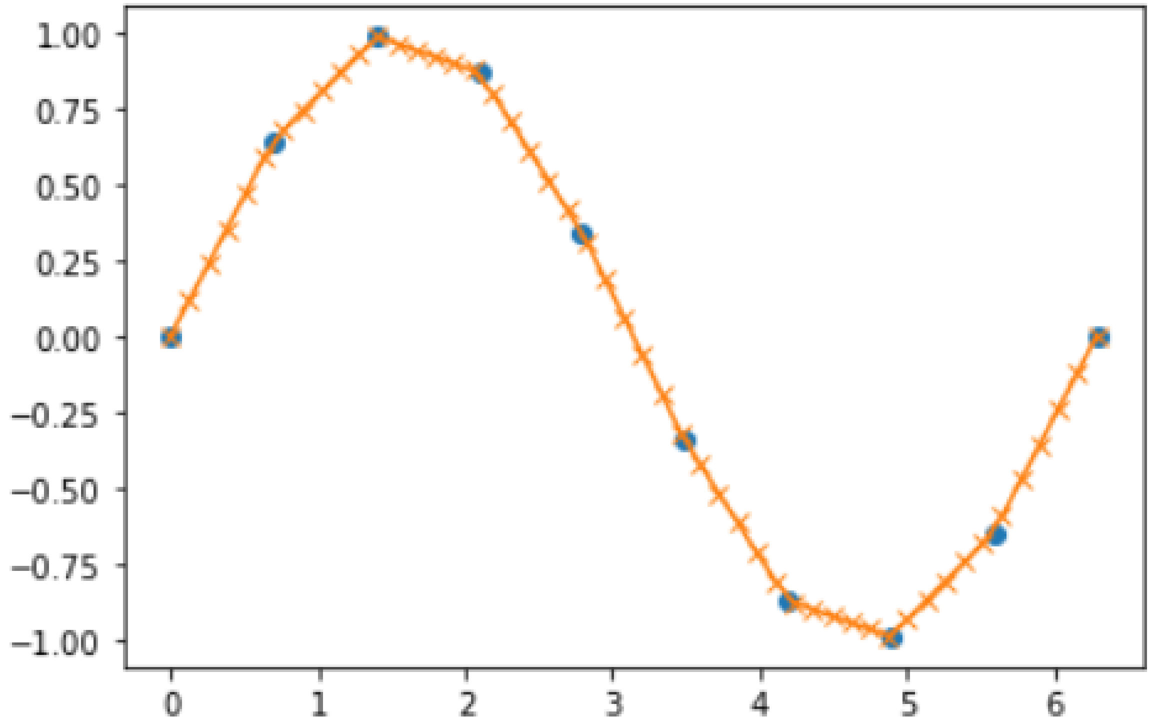


```
x = np.linspace(0, 2*np.pi, 10) # 原始10个点
y = np.sin(x)

xvals = np.linspace(0, 2*np.pi, 50) # 插入50个点
yinterp = np.interp(xvals, x, y)
```

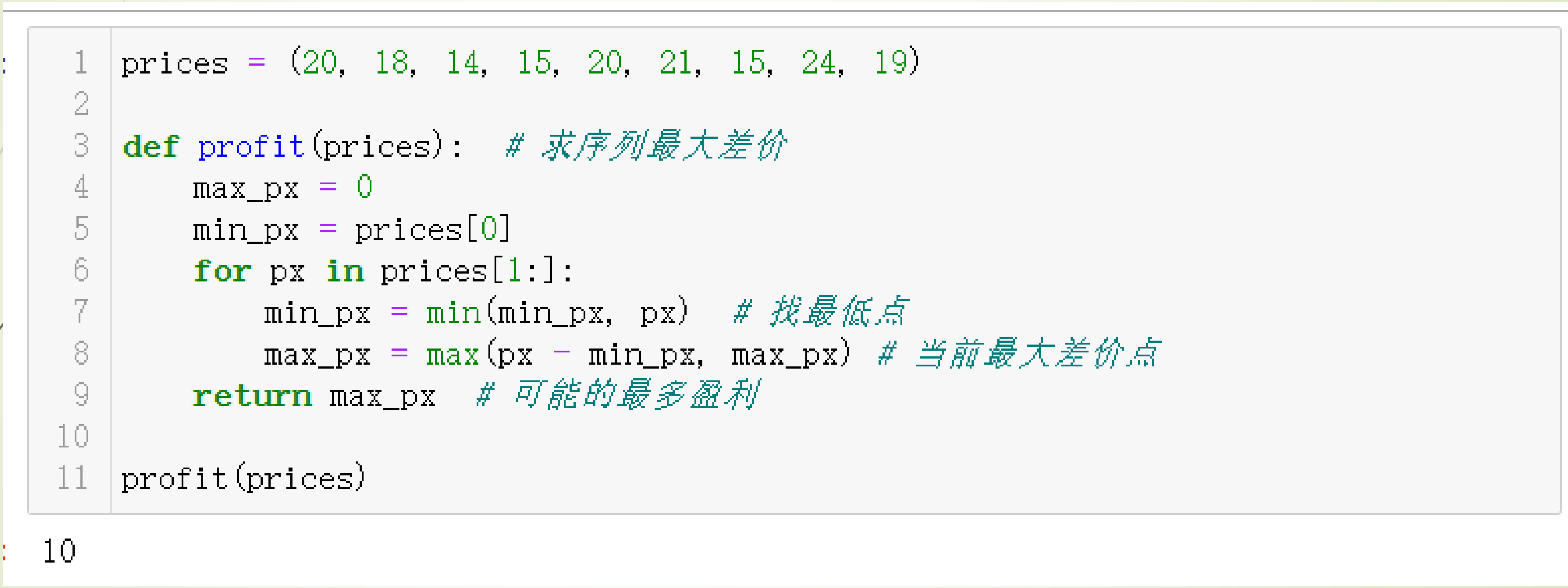


```
import matplotlib.pyplot as plt
plt.plot(x, y, 'o')
plt.plot(xvals, yinterp, '-x')
```

```
[<matplotlib.lines.Line2D at 0x20d2b9b3dc0>]
```



求序列最大差价（示例1）：


```python
prices = (20, 18, 14, 15, 20, 21, 15, 24, 19)

def profit(prices):  # 求序列最大差价
    max_px = 0
    min_px = prices[0]
    for px in prices[1:]:
        min_px = min(min_px, px)  # 找最低点
        max_px = max(px - min_px, max_px) # 当前最大差价点
    return max_px  # 可能的最多盈利

profit(prices)
```

```
10
```

# https://realpython.com/numpy-array-programming/#what-is-vectorization

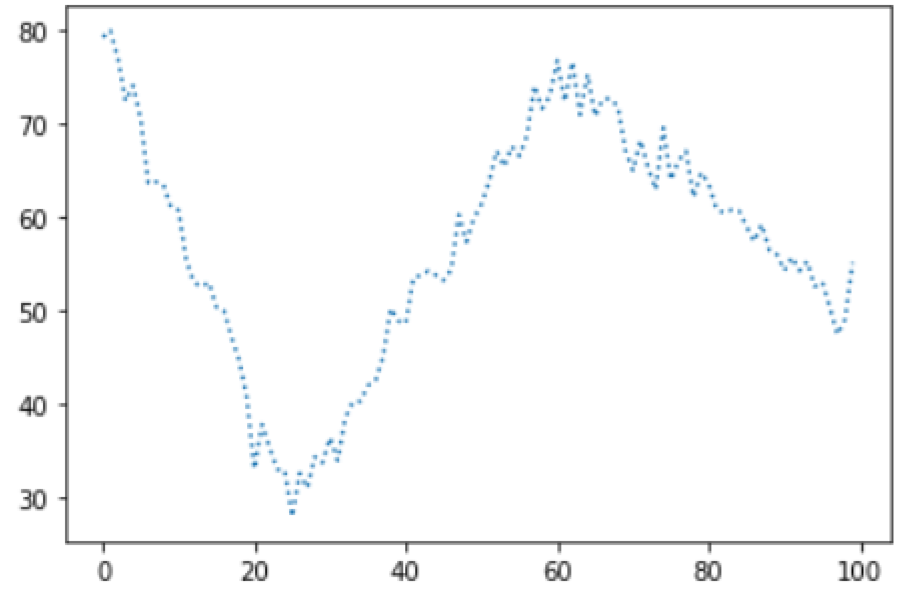
```python
# Create mostly NaN array with a few 'turning points' (local min/max).
prices = np.full(100, fill_value=np.nan)
prices[[0, 25, 60, -1]] = [80., 30., 75., 50.]  # 给几个转折点
prices
```

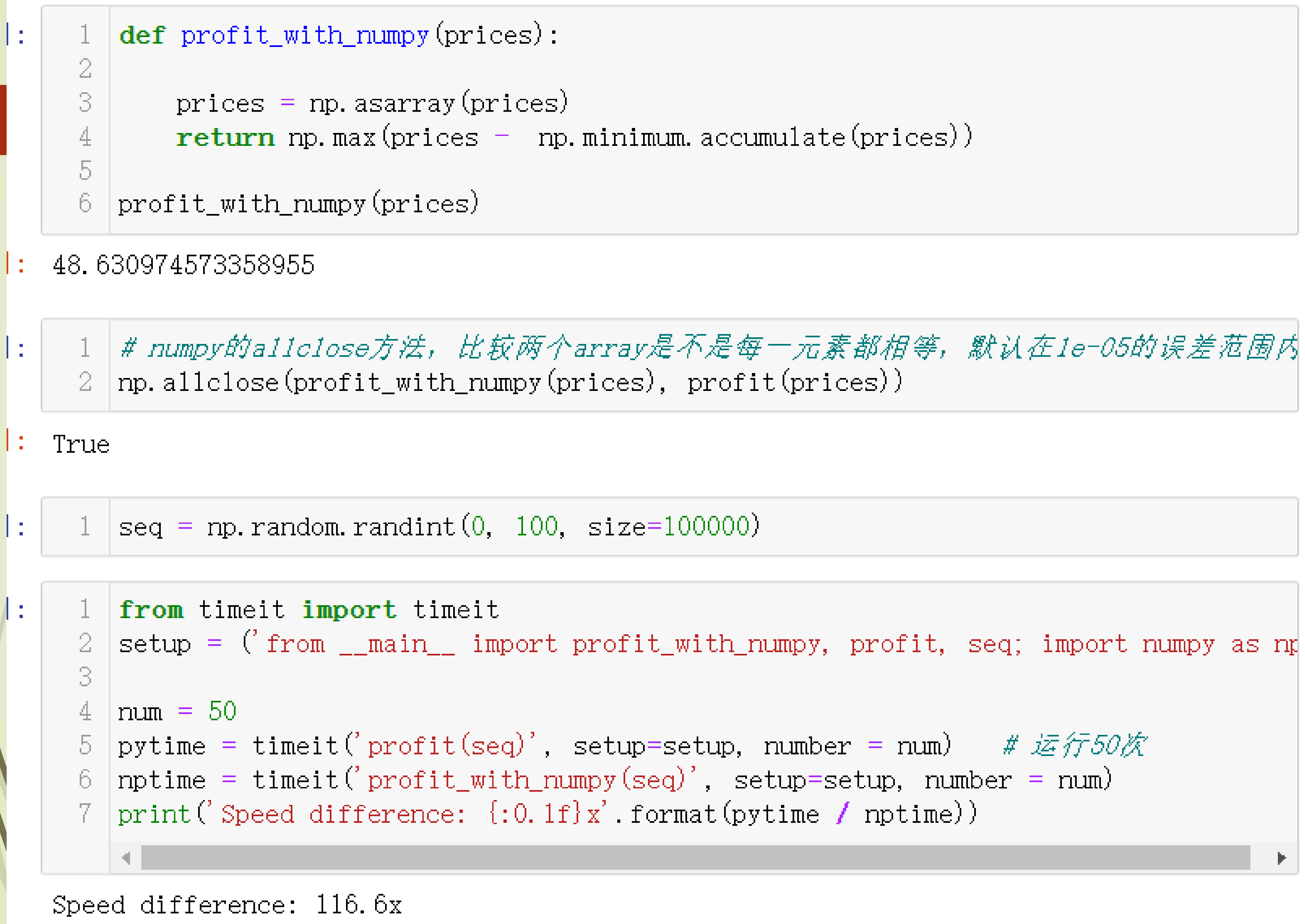
```
array([80., nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, 30.,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, 75., nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, 50.])
```

```python
x = np.arange(len(prices)) # 行坐标
is_valid = ~np.isnan(prices)
prices = np.interp(x=x, xp=x[is_valid], fp=prices[is_valid]) #
#prices += np.random.randn(len(prices)) * 2  # 加噪声
```

```python
plt.plot(prices, linestyle = 'dotted' )
```

```
[<matplotlib.lines.Line2D at 0x20d2c2559a0>]
```





```python
def profit_with_numpy(prices):

    prices = np.asarray(prices)
    return np.max(prices -  np.minimum.accumulate(prices))

profit_with_numpy(prices)
```

```
48.630974573358955
```

```python
# numpy的allclose方法，比较两个array是不是每一元素都相等，默认在1e-05的误差范围内
np.allclose(profit_with_numpy(prices), profit(prices))
```

```
True
```

```python
seq = np.random.randint(0, 100, size=100000)
```

```python
from timeit import timeit
setup = ('from __main__ import profit_with_numpy, profit, seq; import numpy as np

num = 50
pytime = timeit('profit(seq)', setup=setup, number = num)    # 运行50次
nptime = timeit('profit_with_numpy(seq)', setup=setup, number = num)
print('Speed difference: {:0.1f}x'.format(pytime / nptime))
```

```
Speed difference: 116.6x
```

k-means聚类算法简介

- 输入:类的数目k，包含n个文本的特征向量。
- 输出: k个类，使平方误差准则最小。
- 步骤:
- 1)任意选择k个对象作为初始的类中心;
- 2) repeat;
- 3)根据类中对象的平均值，将每个对象(重新)赋给最类似的类;
- 4)更新类的平均值;
- 5) until不再发生变化。

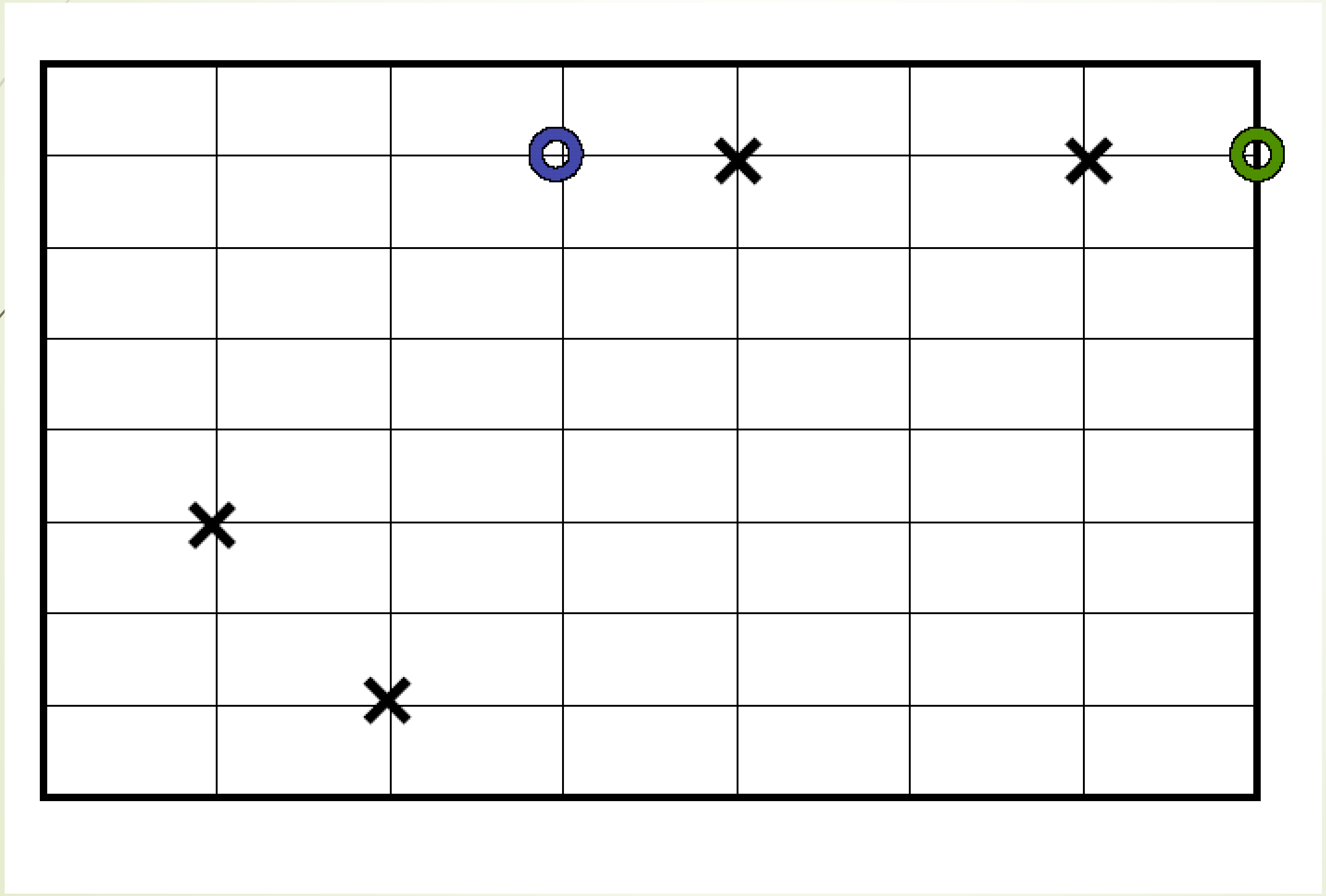
K-均值举例

- 将n个向量分到k个类别中去
  - $x_1=(2,1)$ $x_2=(1,3)$ $x_3=(6,7)$ $x_4=(4,7)$
- 选择k个初始中心
  - $f_1=(4,3)$ $f_2=(5,5)$
- 计算两项距离
- 计算均值

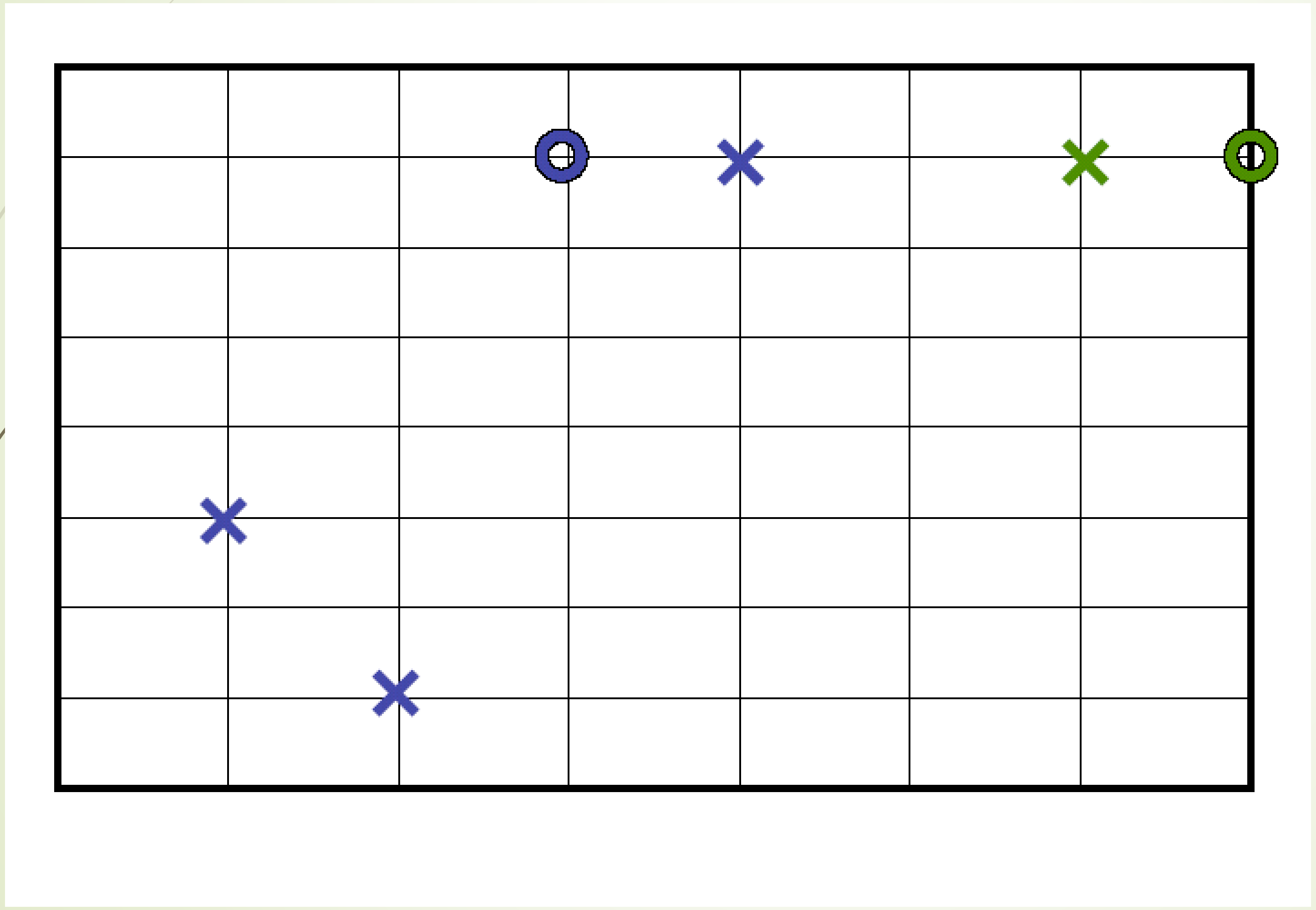
$$d(x_i,x_j) = \sqrt{\sum_{k=1}^{n}(x_{i_k} - x_{j_k})^2}$$

$$\mu(x_1,\ldots,x_n) = \left(\frac{\sum_{i=1}^{n} x_{i_1}}{n},\ldots,\frac{\sum_{i=1}^{n} x_{i_m}}{n}\right)$$

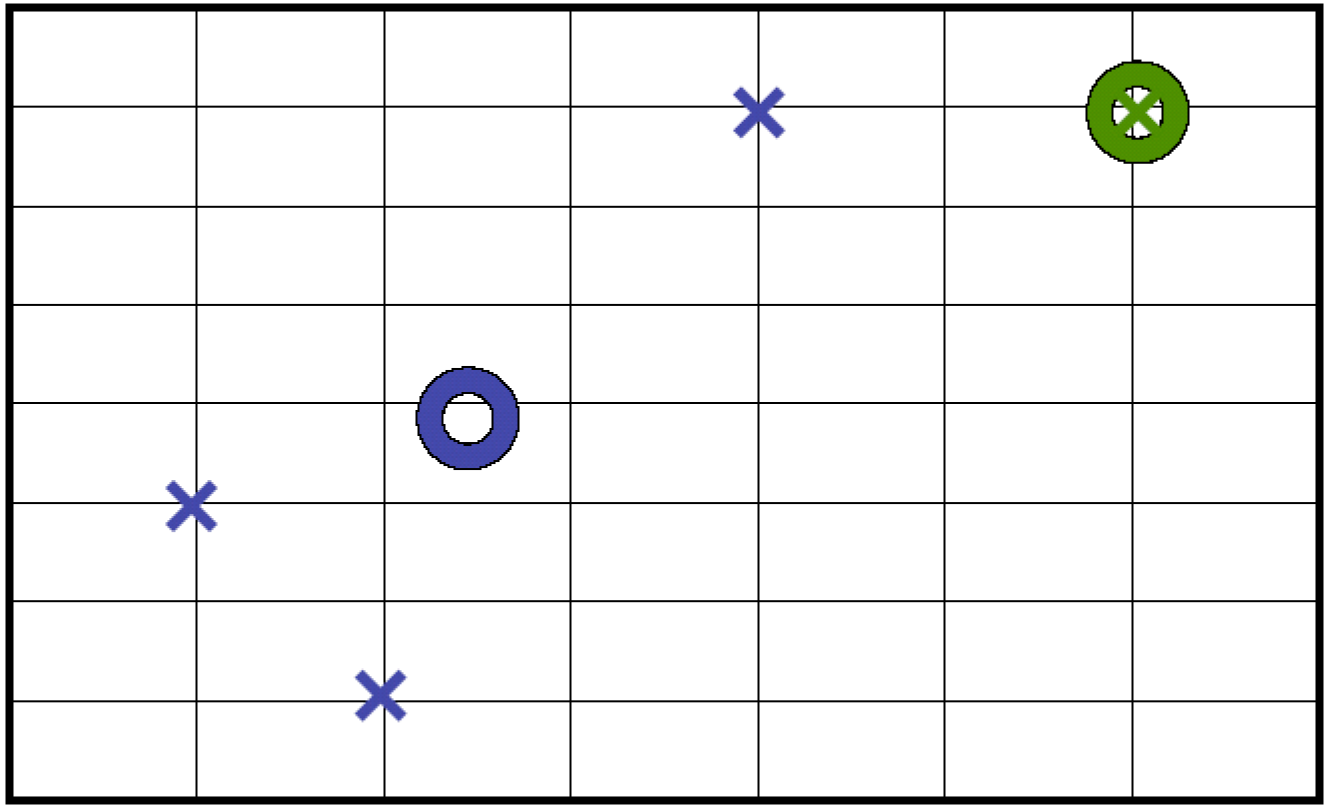
随机两个kernel



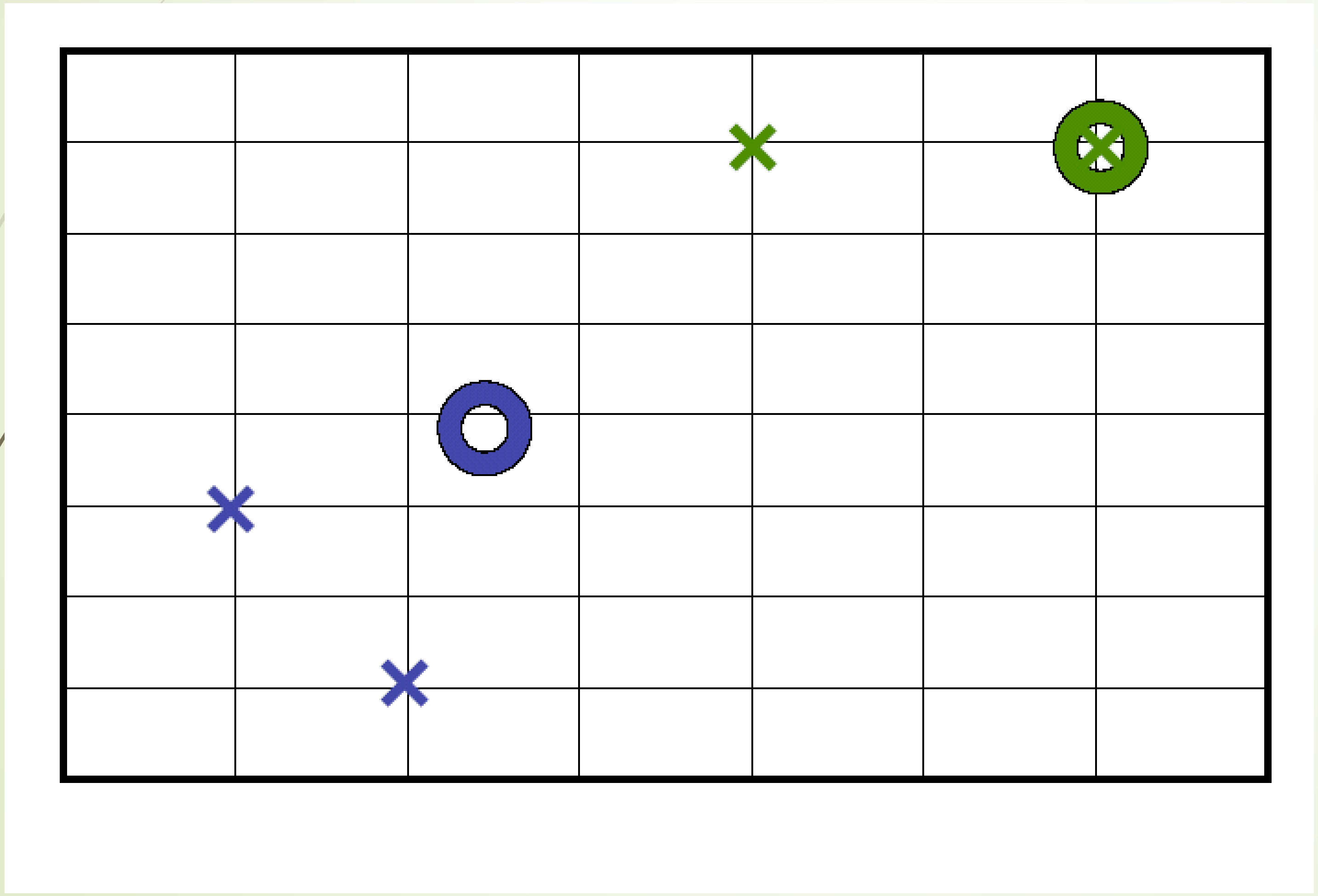
完成按距离的划分



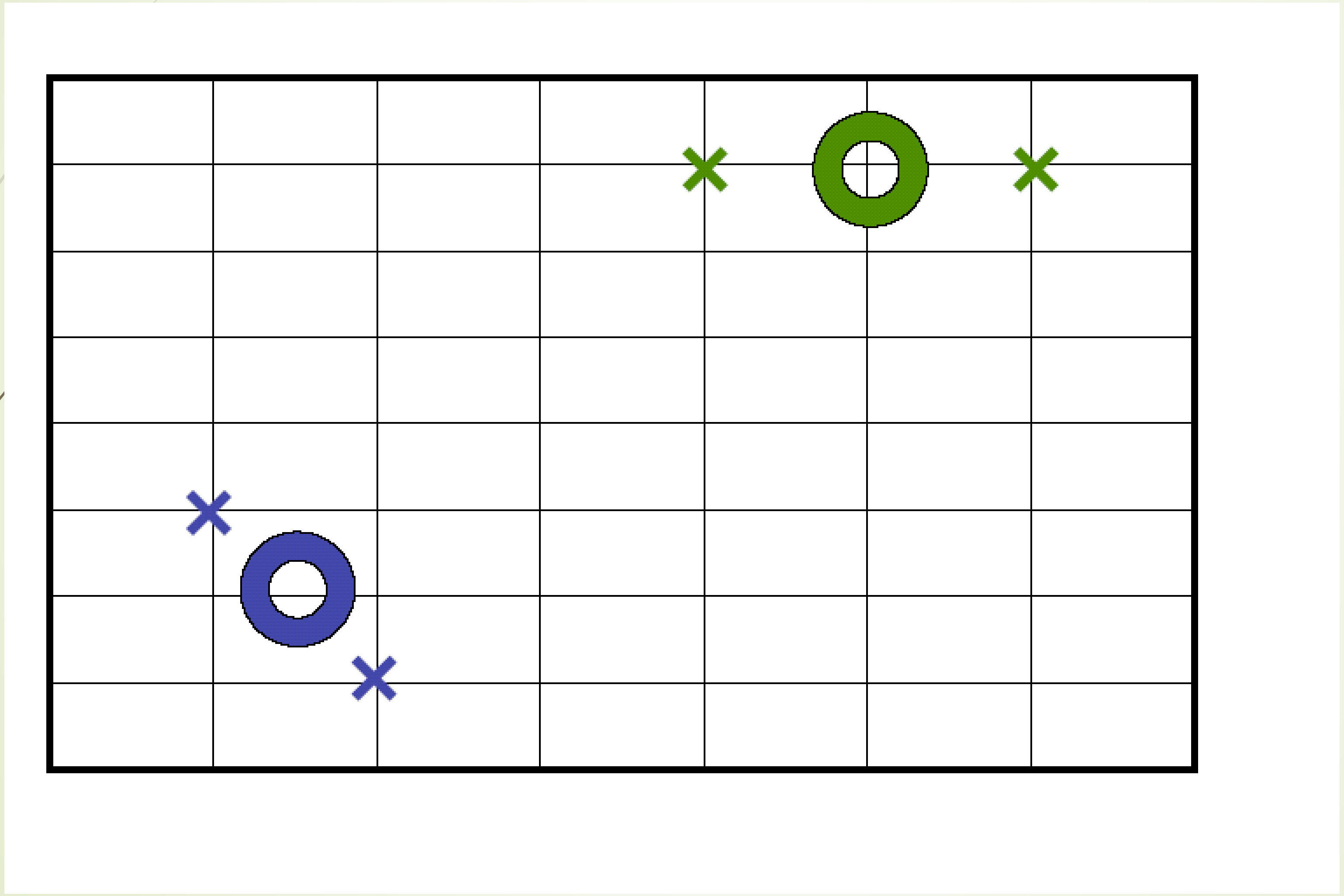
根据划分计算出每个类新的kernel



按新的kernel重新对样本进行划分



迭代到收斂

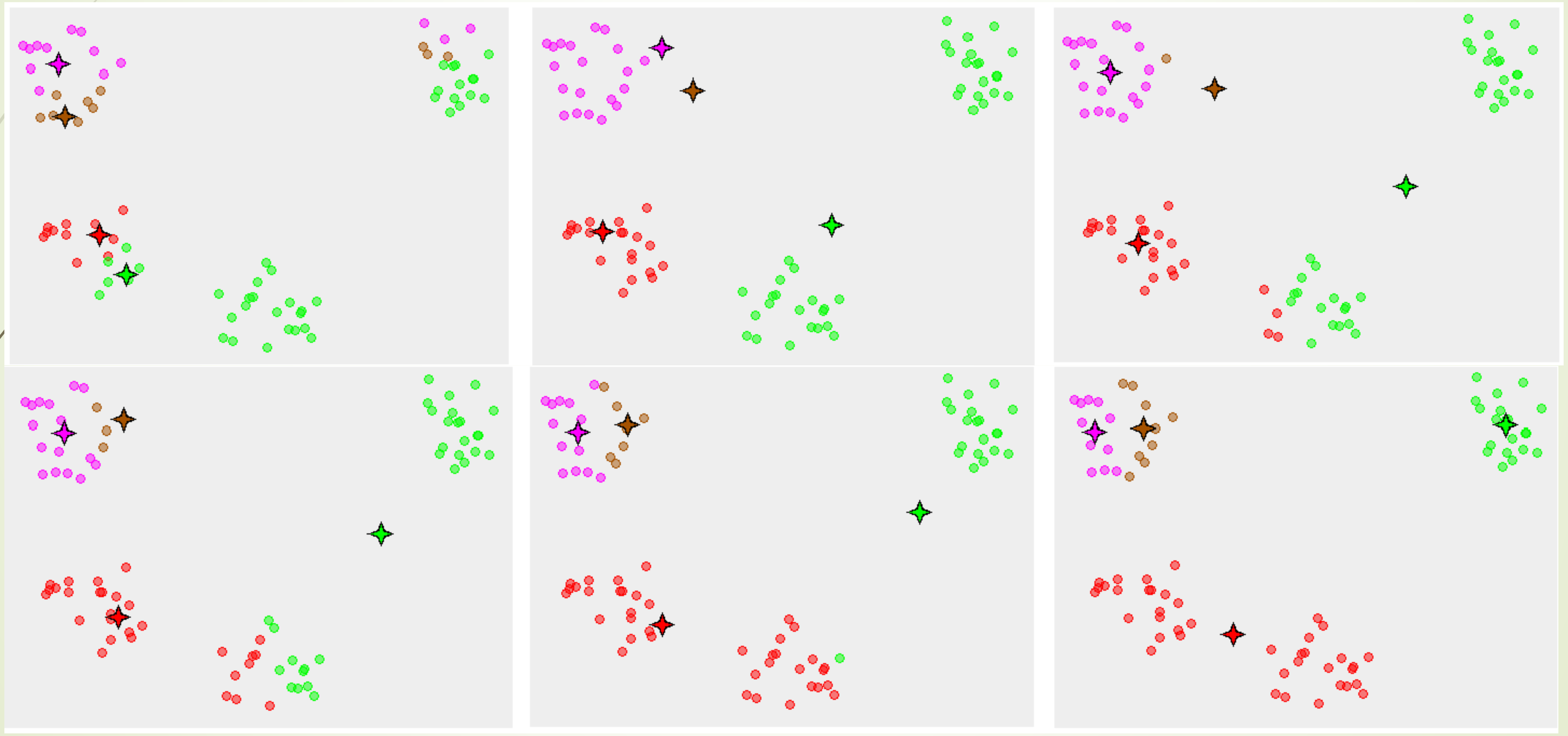


K-均值算法分析

- 算法复杂度为O (kln) ，其中l为迭代次数， n为文档个数， k为类别个数
- K-均值算法最后一定是可以收敛的
- 该算法本质上是一种贪心算法。
  - 可以保证局部最优，但是很难保证全局最优。
- 需要预先指定k值和初始划分

K-means convergence to a local minimum

— From Wikipedia



K-means的物理意义：

- VQ下平方误差最小
- 能否采用其他的误差方案？

K-means 的改进

- 确定K
  - 对于不同的K都尝试聚类，取效果最好的
- 确定初始种子
  - 排除明显是“噪声”的文档向量
  - 尝试多种初始向量的组合，取效果最好的
  - 通过其他方法（如层次聚类）确定初始文档向量

聚类的质量评价

- 指标：纯度（Purity）和F值（F-measure）
- 标准答案：一般是人工分好类的文档集合

纯度

- 对于聚类后形成的任意类别r，聚类的纯度定义为

$$P(S_r) = \frac{1}{n_r} \max(n_r^i)$$

- 整个聚类结果的纯度定义为

$$\text{Purity} = \sum_{r=1}^{k} \frac{n_r}{n} P(S_r)$$

- n_r^i：属于预定义类i且被分配到第r个聚类的文档个数
- n_r：第r个聚类类别中的文档个数

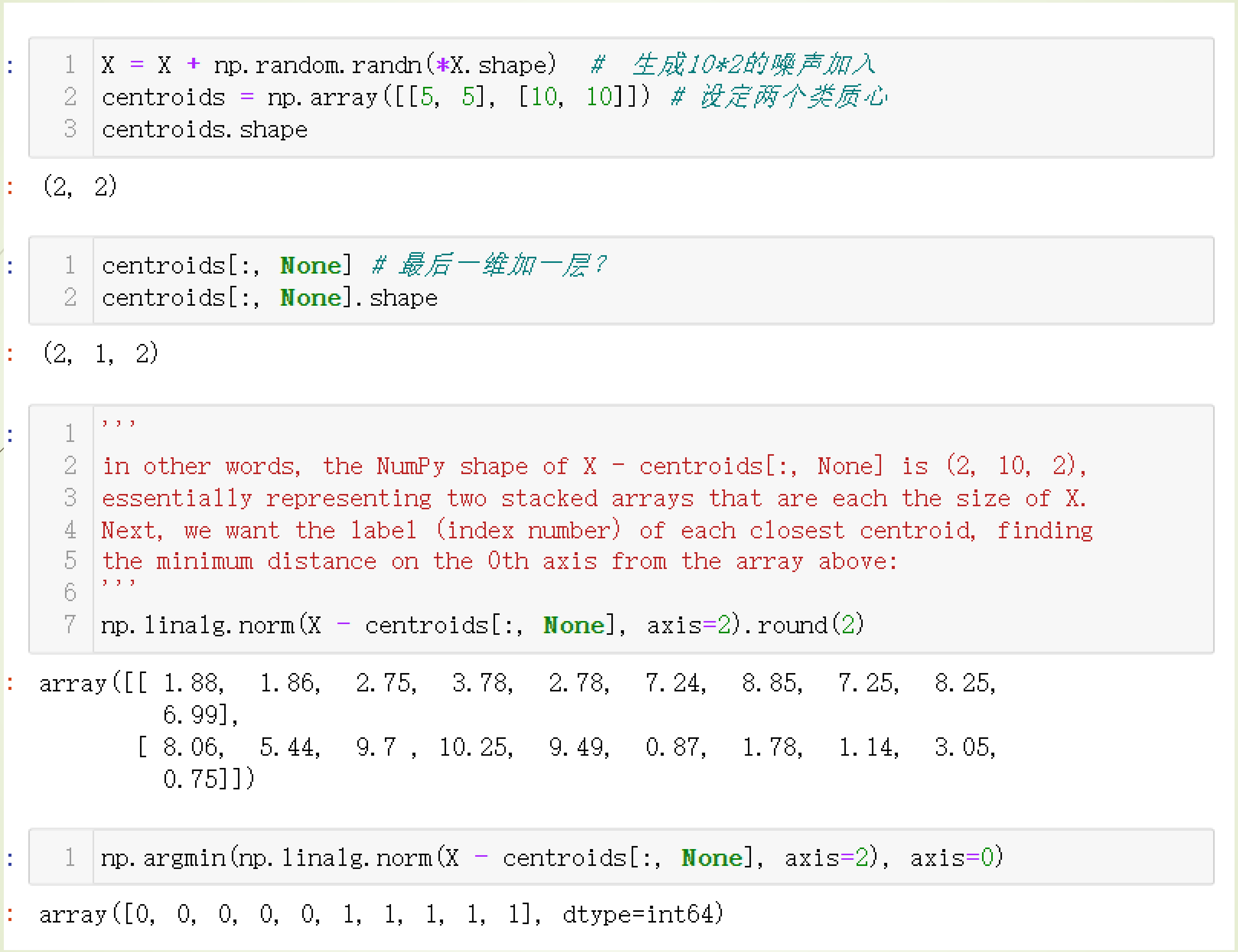
F值

- F值：准确率（precision）和召回率（recall）的调和平均数
- $\mathrm{precision(i, r)} = n_r^i / n_r$
- $\mathrm{recall(i, r)} = n_r^i / n_i$
- n_r^i：属于预定义类i且被分配到第r个聚类的文档个数
- n_r：第r个聚类类别中的文档个数
- n_i：预定义类别i中的文档个数

F值

- 聚类r和类别i之间的f值计算如下：
- $f(i,r) = \frac{2 \times recall(i,r) \times precision(i,r)}{precision(i,r) + recall(i,r)}$

- 最终聚类结果的评价函数为
- $F = \sum_i \frac{n_i}{n} \max\{f(i,r)\}$，$n$是所有文档的个数



```python
X = X + np.random.randn(*X.shape)   #  生成10*2的噪声加入
centroids = np.array([[5, 5], [10, 10]]) # 设定两个类质心
centroids.shape
```

```
(2, 2)
```

```python
centroids[:, None] # 最后一维加一层？
centroids[:, None].shape
```

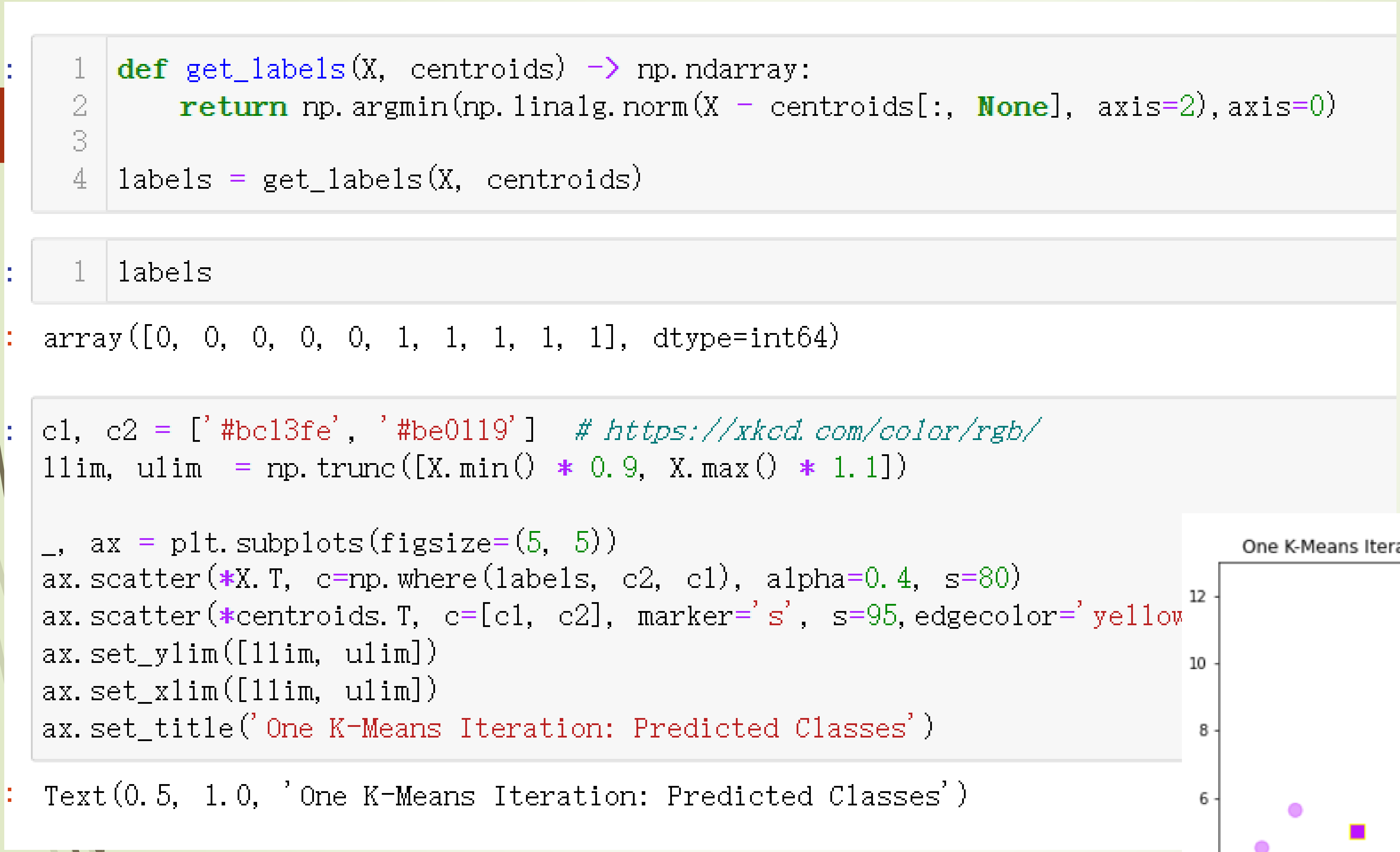
```
(2, 1, 2)
```

```python
'''
in other words, the NumPy shape of X - centroids[:, None] is (2, 10, 2),
essentially representing two stacked arrays that are each the size of X.
Next, we want the label (index number) of each closest centroid, finding
the minimum distance on the 0th axis from the array above:
'''
np.linalg.norm(X - centroids[:, None], axis=2).round(2)
```

```
array([[ 1.88,  1.86,  2.75,  3.78,  2.78,  7.24,  8.85,  7.25,  8.25,
         6.99],
       [ 8.06,  5.44,  9.7 , 10.25,  9.49,  0.87,  1.78,  1.14,  3.05,
         0.75]])
```

```python
np.argmin(np.linalg.norm(X - centroids[:, None], axis=2), axis=0)
```

```
array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1], dtype=int64)
```



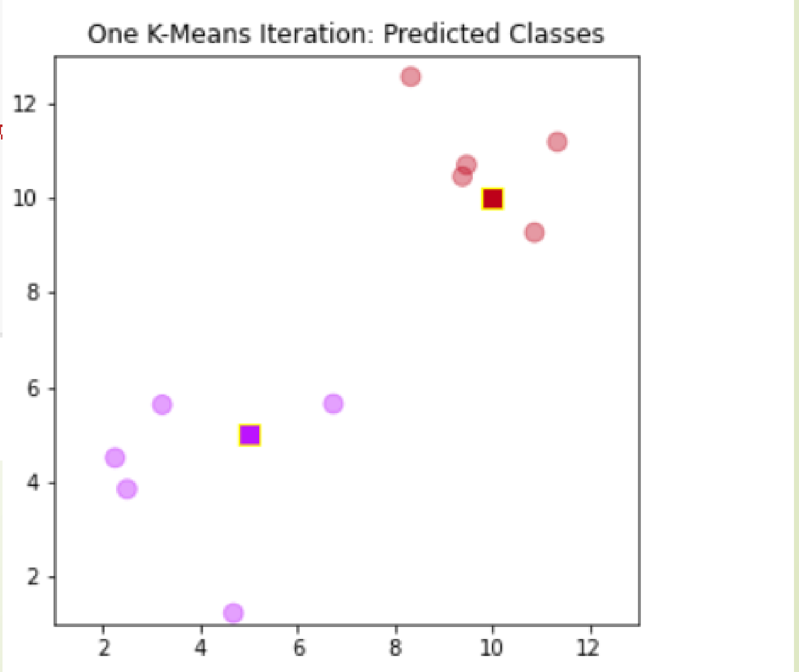
```python
def get_labels(X, centroids) -> np.ndarray:
    return np.argmin(np.linalg.norm(X - centroids[:, None], axis=2),axis=0)

labels = get_labels(X, centroids)
```

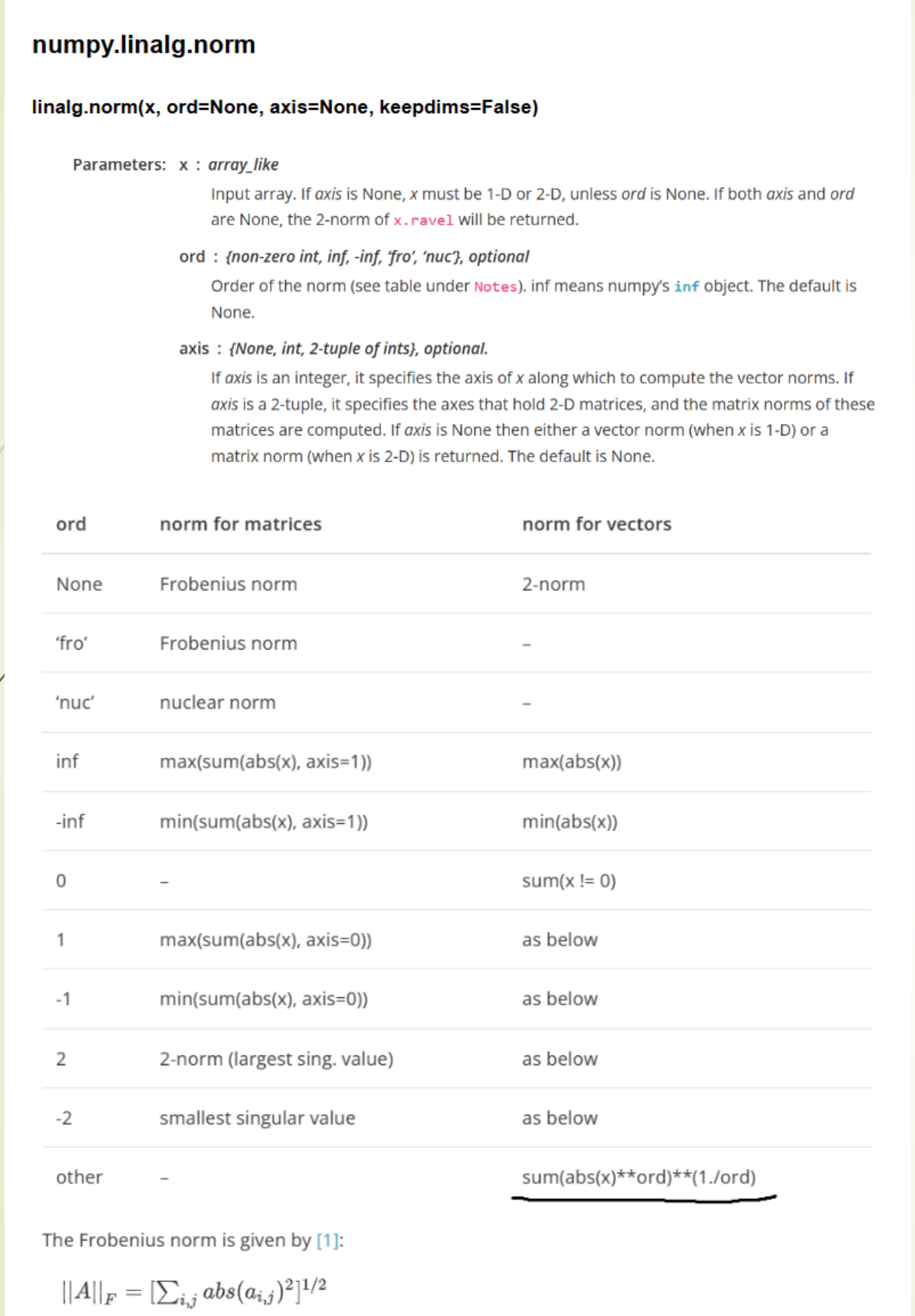
```python
labels
```

```
array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1], dtype=int64)
```

```python
c1, c2 = ['#bc13fe', '#be0119']  # https://xkcd.com/color/rgb/
llim, ulim  = np.trunc([X.min() * 0.9, X.max() * 1.1])

_, ax = plt.subplots(figsize=(5, 5))
ax.scatter(*X.T, c=np.where(labels, c2, c1), alpha=0.4, s=80)
ax.scatter(*centroids.T, c=[c1, c2], marker='s', s=95,edgecolor='yellow
ax.set_ylim([llim, ulim])
ax.set_xlim([llim, ulim])
ax.set_title('One K-Means Iteration: Predicted Classes')
```

```
Text(0.5, 1.0, 'One K-Means Iteration: Predicted Classes')
```





numpy.linalg.norm

linalg.norm(x, ord=None, axis=None, keepdims=False)

Parameters: **x** : *array_like*

Input array. If *axis* is None, *x* must be 1-D or 2-D, unless *ord* is None. If both *axis* and *ord* are None, the 2-norm of `x.ravel` will be returned.

ord : *{non-zero int, inf, -inf, 'fro', 'nuc'}, optional*

Order of the norm (see table under `Notes`). inf means numpy's `inf` object. The default is None.

axis : *{None, int, 2-tuple of ints}, optional.*

If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned. The default is None.

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
'nuc'	nuclear norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

The Frobenius norm is given by [1]:

$$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$$

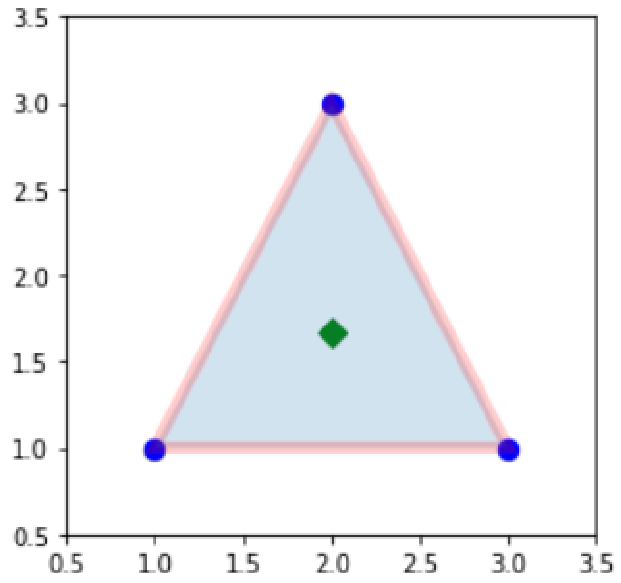
```
tri = np.array([[1,  1],
                [3,  1],
                [2,  3]])    #
```

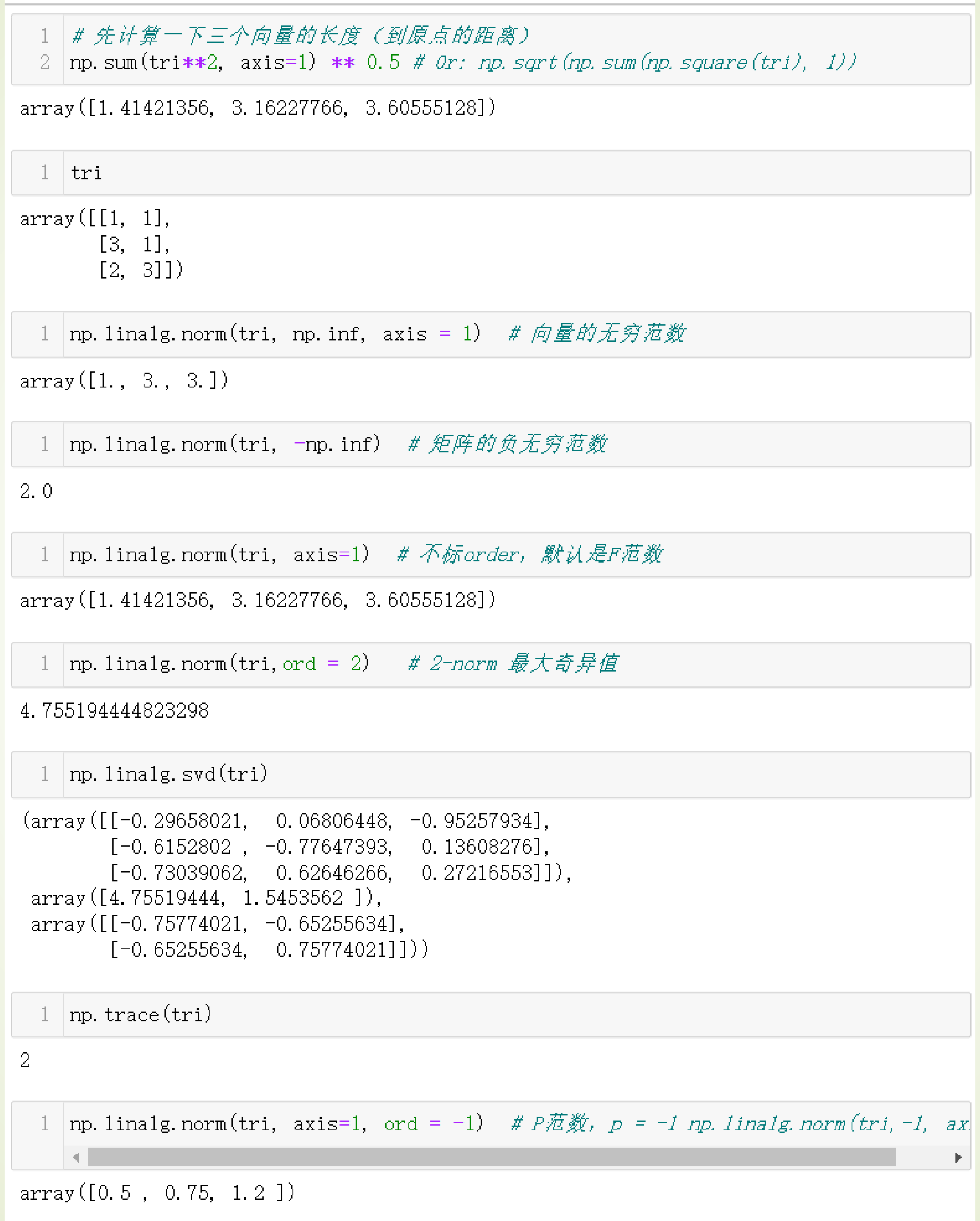
```
centroid = tri.mean(axis=0)
centroid
```

```
array([2.        , 1.66666667])
```

```
trishape = plt.Polygon(tri, edgecolor='r', alpha=0.2, lw=5)
_, ax = plt.subplots(figsize=(4, 4))
ax.add_patch(trishape)
ax.set_ylim([.5, 3.5])
ax.set_xlim([.5, 3.5])
ax.scatter(*centroid, color='g', marker='D', s=70)
ax.scatter(*tri.T, color='b',  s=70)
```

```
<matplotlib.collections.PathCollection at 0x20d2c239fd0>
```





```
# 先计算一下三个向量的长度（到原点的距离）
np.sum(tri**2, axis=1) ** 0.5 # Or: np.sqrt(np.sum(np.square(tri), 1))
```

```
array([1.41421356, 3.16227766, 3.60555128])
```

```
tri
```

```
array([[1, 1],
       [3, 1],
       [2, 3]])
```

```
np.linalg.norm(tri, np.inf, axis = 1)  # 向量的无穷范数
```

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

```
np.linalg.norm(tri, -np.inf)  # 矩阵的负无穷范数
```

```
2.0
```

```
np.linalg.norm(tri, axis=1)  # 不标order, 默认是F范数
```

```
array([1.41421356, 3.16227766, 3.60555128])
```

```
np.linalg.norm(tri,ord = 2)   # 2-norm 最大奇异值
```

```
4.755194444823298
```

```
np.linalg.svd(tri)
```

```
(array([[-0.29658021,  0.06806448, -0.95257934],
        [-0.6152802 , -0.77647393,  0.13608276],
        [-0.73039062,  0.62646266,  0.27216553]]),
 array([4.75519444, 1.5453562 ]),
 array([[-0.75774021, -0.65255634],
        [-0.65255634,  0.75774021]]))
```

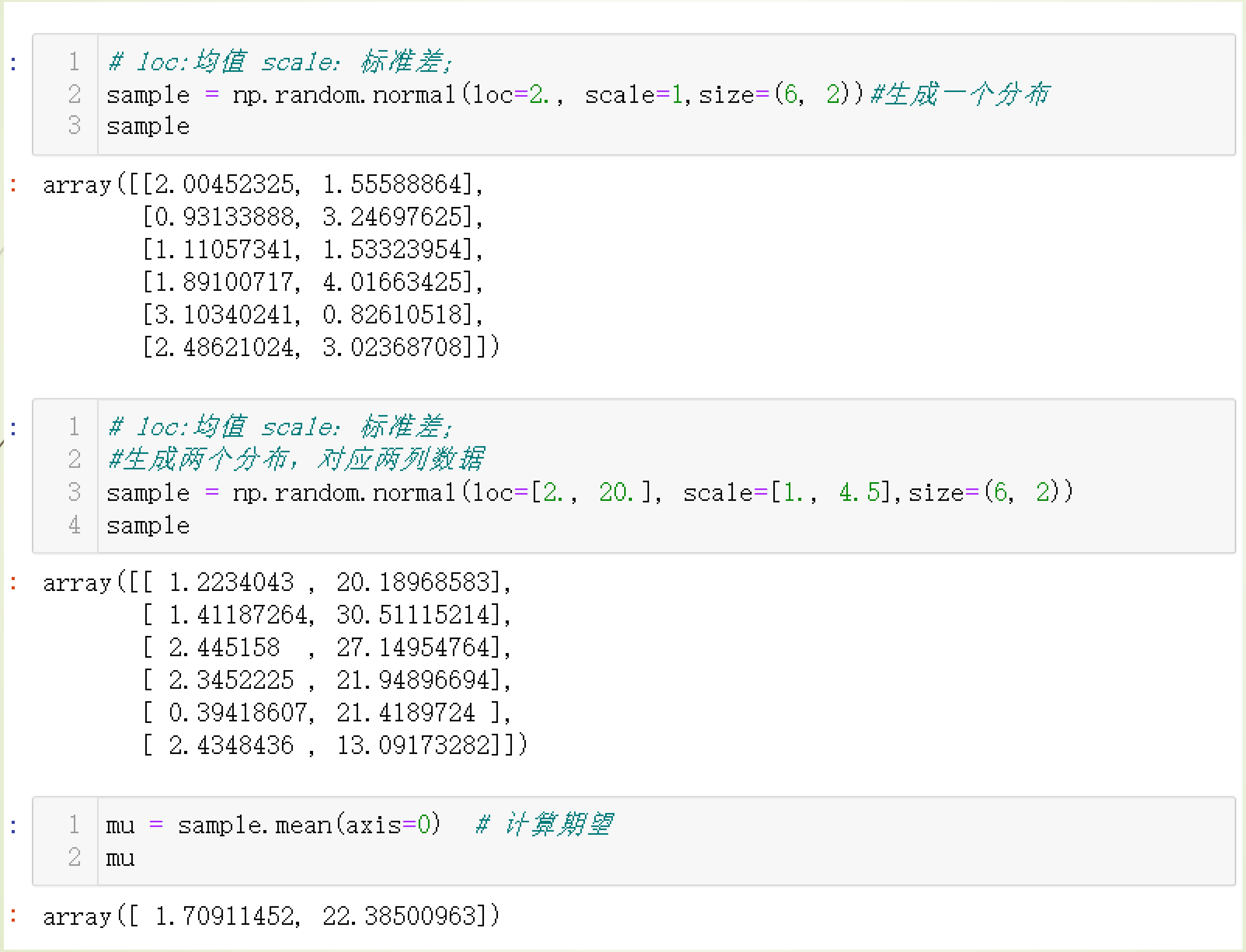
```
np.trace(tri)
```

```
2
```

```
np.linalg.norm(tri, axis=1, ord = -1)  # P范数, p = -1 np.linalg.norm(tri,-1, ax
```

```
array([0.5 , 0.75, 1.2 ])
```

数据标准化：



```
# loc:均值 scale: 标准差;
sample = np.random.normal(loc=2., scale=1,size=(6, 2))#生成一个分布
sample
```

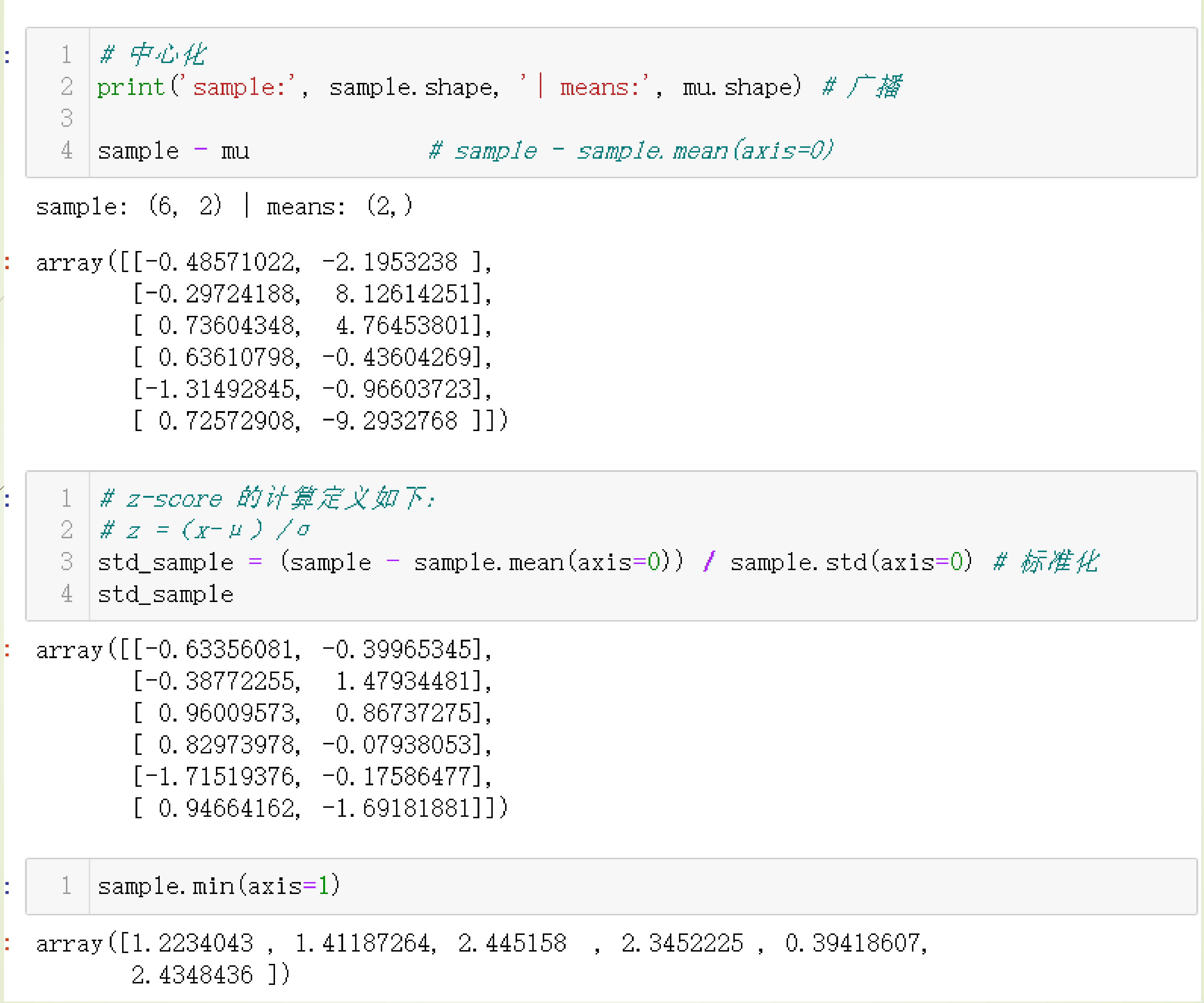
```
array([[2.00452325, 1.55588864],
       [0.93133888, 3.24697625],
       [1.11057341, 1.53323954],
       [1.89100717, 4.01663425],
       [3.10340241, 0.82610518],
       [2.48621024, 3.02368708]])
```

```
# loc:均值 scale: 标准差;
#生成两个分布，对应两列数据
sample = np.random.normal(loc=[2., 20.], scale=[1., 4.5],size=(6, 2))
sample
```

```
array([[ 1.2234043 , 20.18968583],
       [ 1.41187264, 30.51115214],
       [ 2.445158  , 27.14954764],
       [ 2.3452225 , 21.94896694],
       [ 0.39418607, 21.4189724 ],
       [ 2.4348436 , 13.09173282]])
```

```
mu = sample.mean(axis=0)  # 计算期望
mu
```

```
array([ 1.70911452, 22.38500963])
```



```python
# 中心化
print('sample:', sample.shape, '| means:', mu.shape) # 广播

sample - mu                   # sample - sample.mean(axis=0)
```

```
sample: (6, 2) | means: (2,)
```

```
array([[-0.48571022, -2.1953238 ],
       [-0.29724188,  8.12614251],
       [ 0.73604348,  4.76453801],
       [ 0.63610798, -0.43604269],
       [-1.31492845, -0.96603723],
       [ 0.72572908, -9.2932768 ]])
```

```python
# z-score 的计算定义如下:
# z = (x-μ) /σ
std_sample = (sample - sample.mean(axis=0)) / sample.std(axis=0) # 标准化
std_sample
```

```
array([[-0.63356081, -0.39965345],
       [-0.38772255,  1.47934481],
       [ 0.96009573,  0.86737275],
       [ 0.82973978, -0.07938053],
       [-1.71519376, -0.17586477],
       [ 0.94664162, -1.69181881]])
```

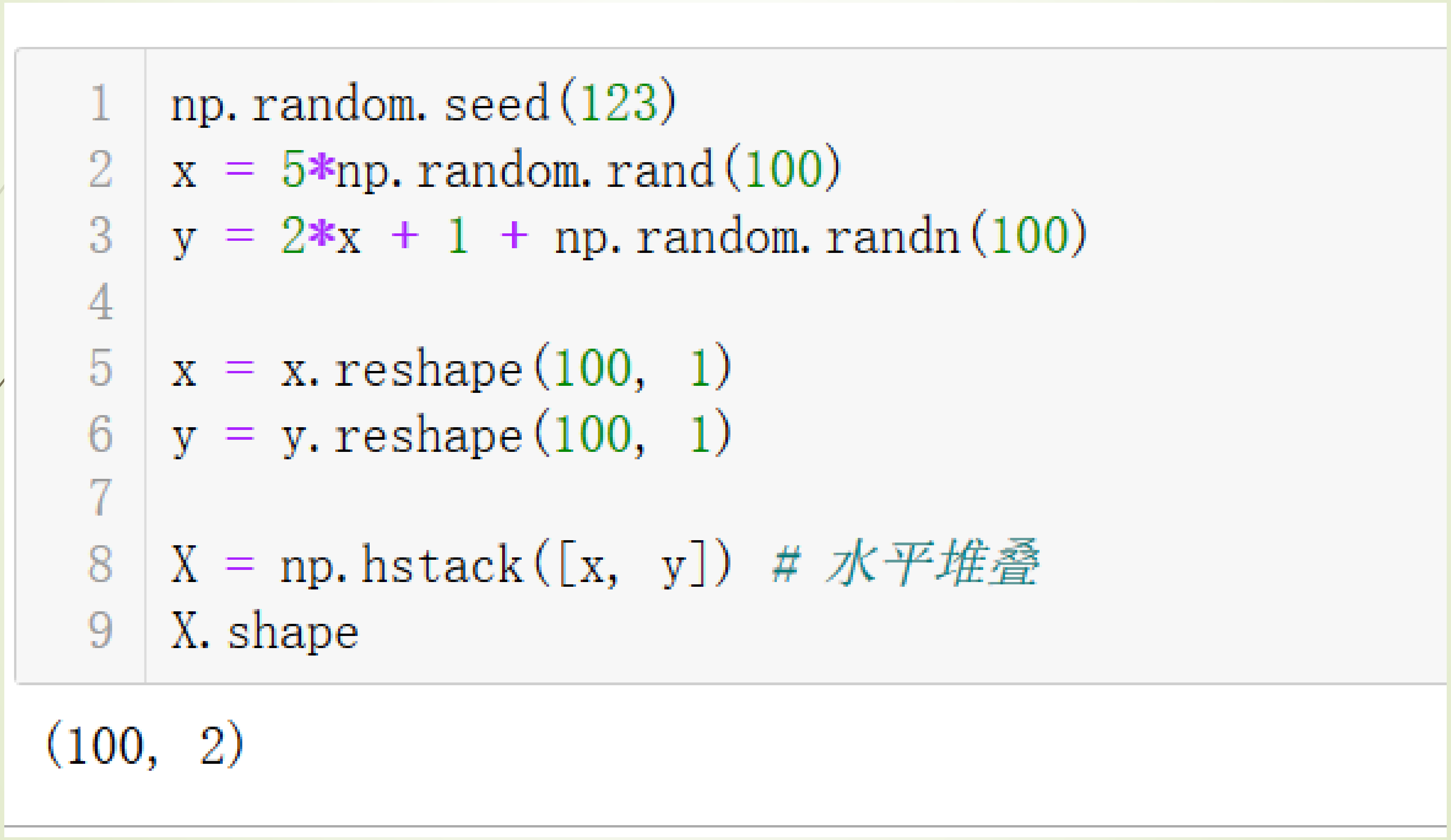
```python
sample.min(axis=1)
```

```
array([1.2234043 , 1.41187264, 2.445158  , 2.3452225 , 0.39418607,
       2.4348436 ])
```

矩阵的特征值与特征向量

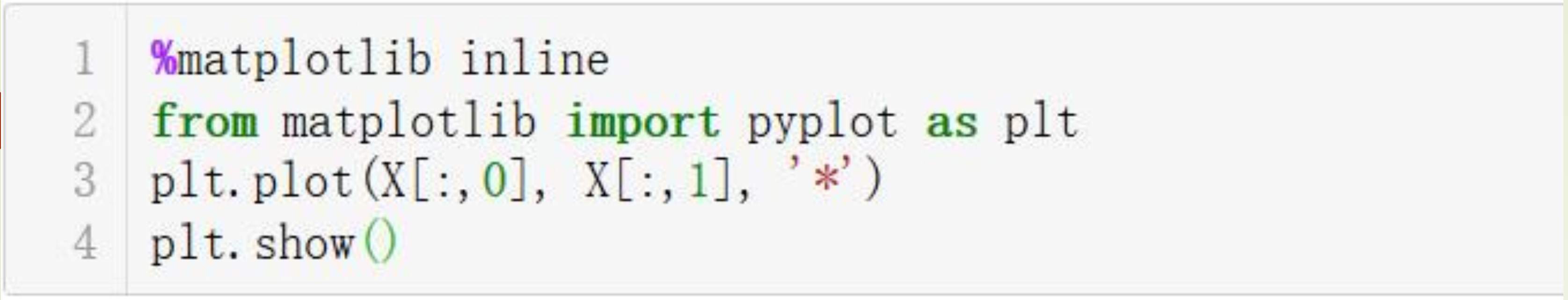
- 求特征值
- 特征向量的物理意义
- 数据降维

生成二维数据：

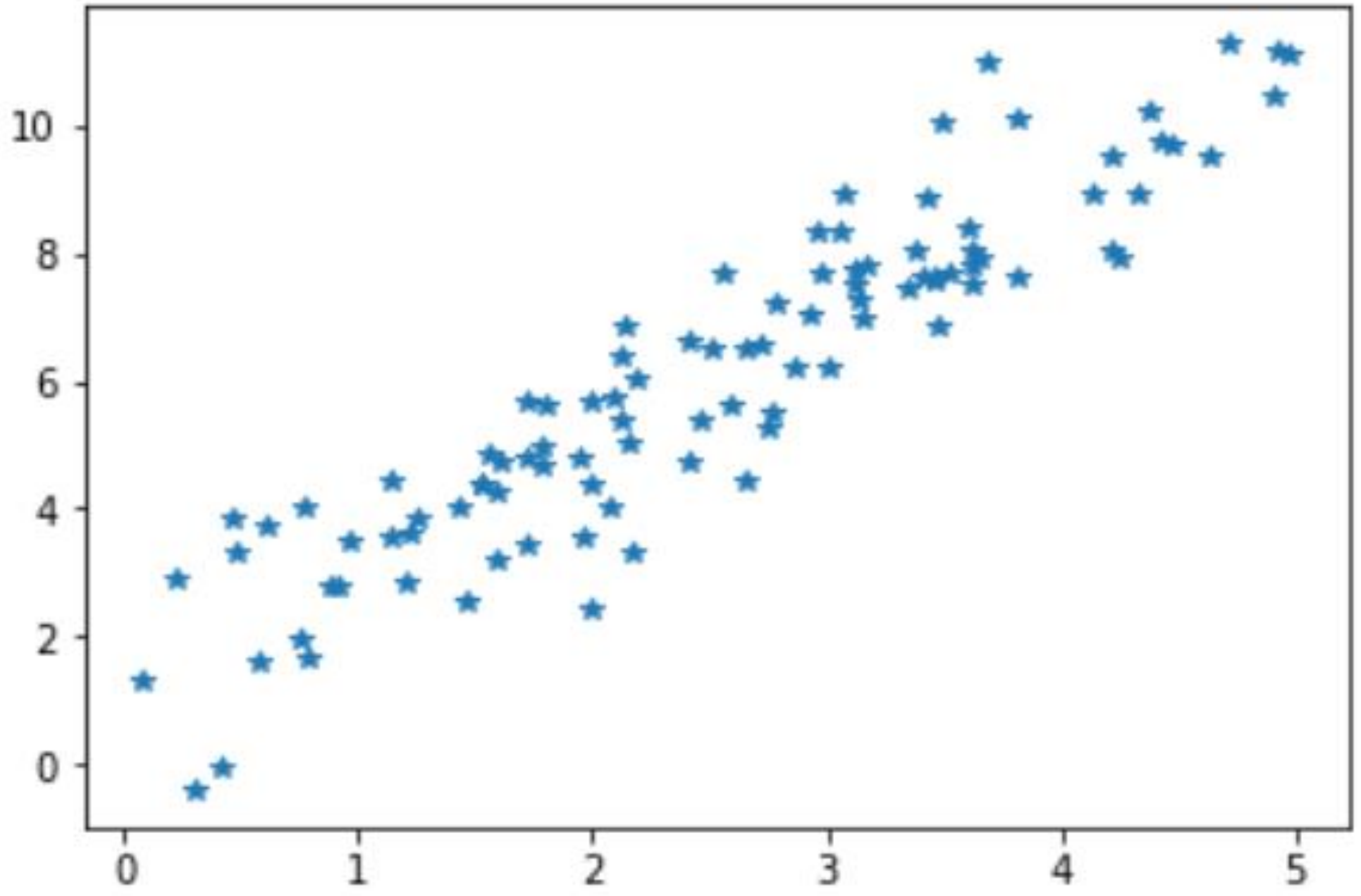


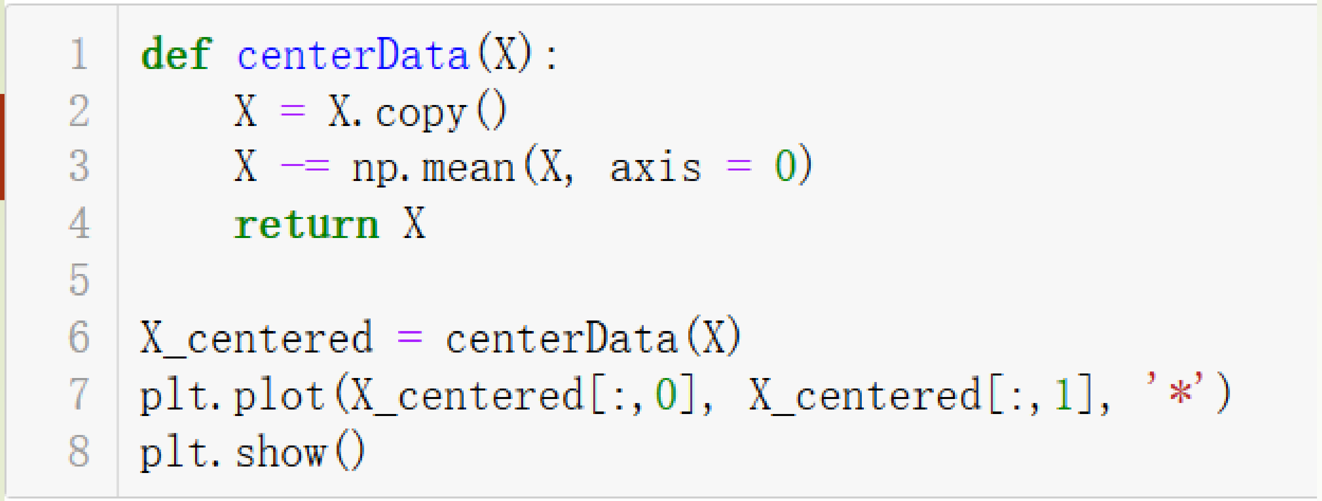
```
np.random.seed(123)
x = 5*np.random.rand(100)
y = 2*x + 1 + np.random.randn(100)

x = x.reshape(100, 1)
y = y.reshape(100, 1)

X = np.hstack([x, y]) # 水平堆叠
X.shape
```

```
(100, 2)
```

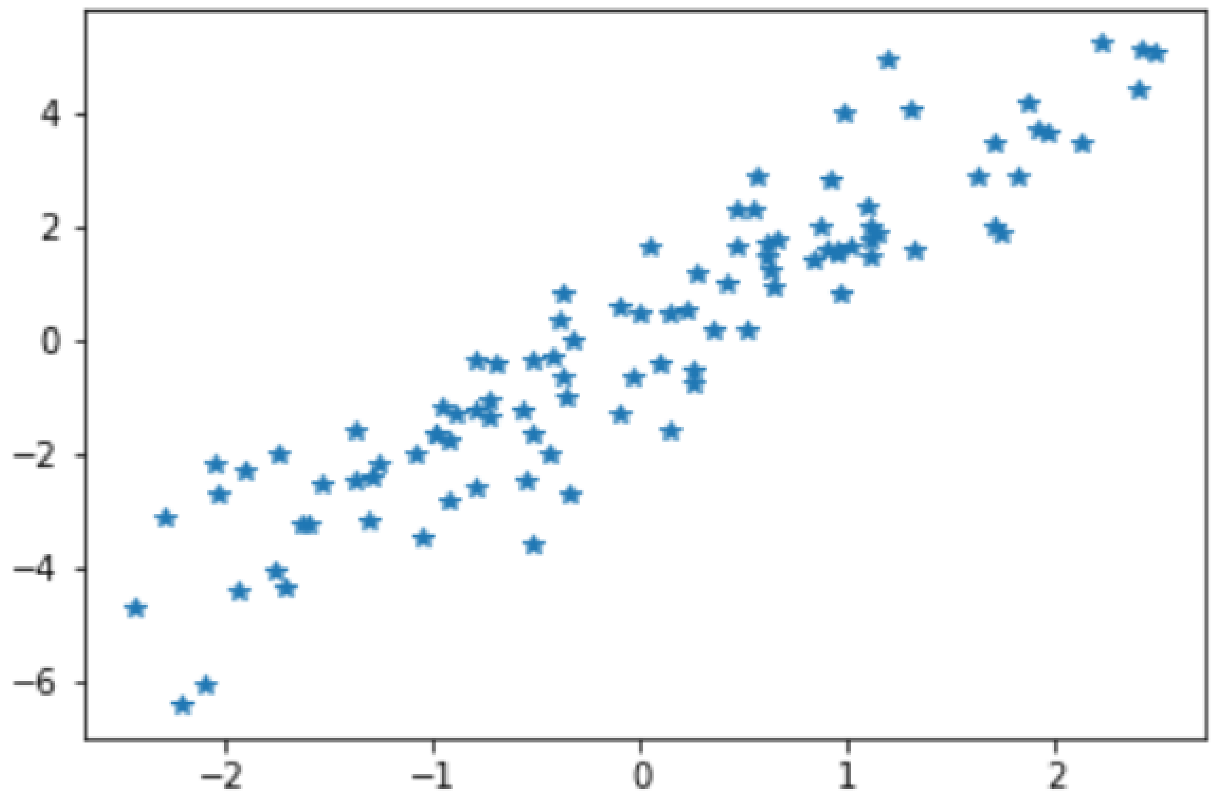


```python
%matplotlib inline
from matplotlib import pyplot as plt
plt.plot(X[:,0], X[:,1], '*')
plt.show()
```





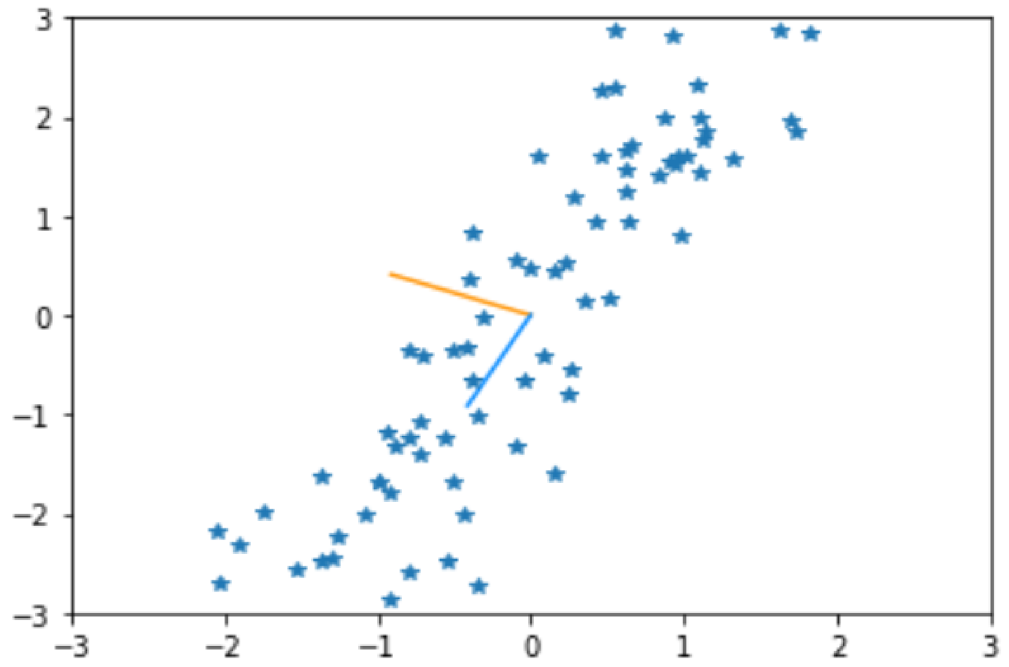
```
def centerData(X):
    X = X.copy()
    X -= np.mean(X, axis = 0)
    return X

X_centered = centerData(X)
plt.plot(X_centered[:,0], X_centered[:,1], '*')
plt.show()
```

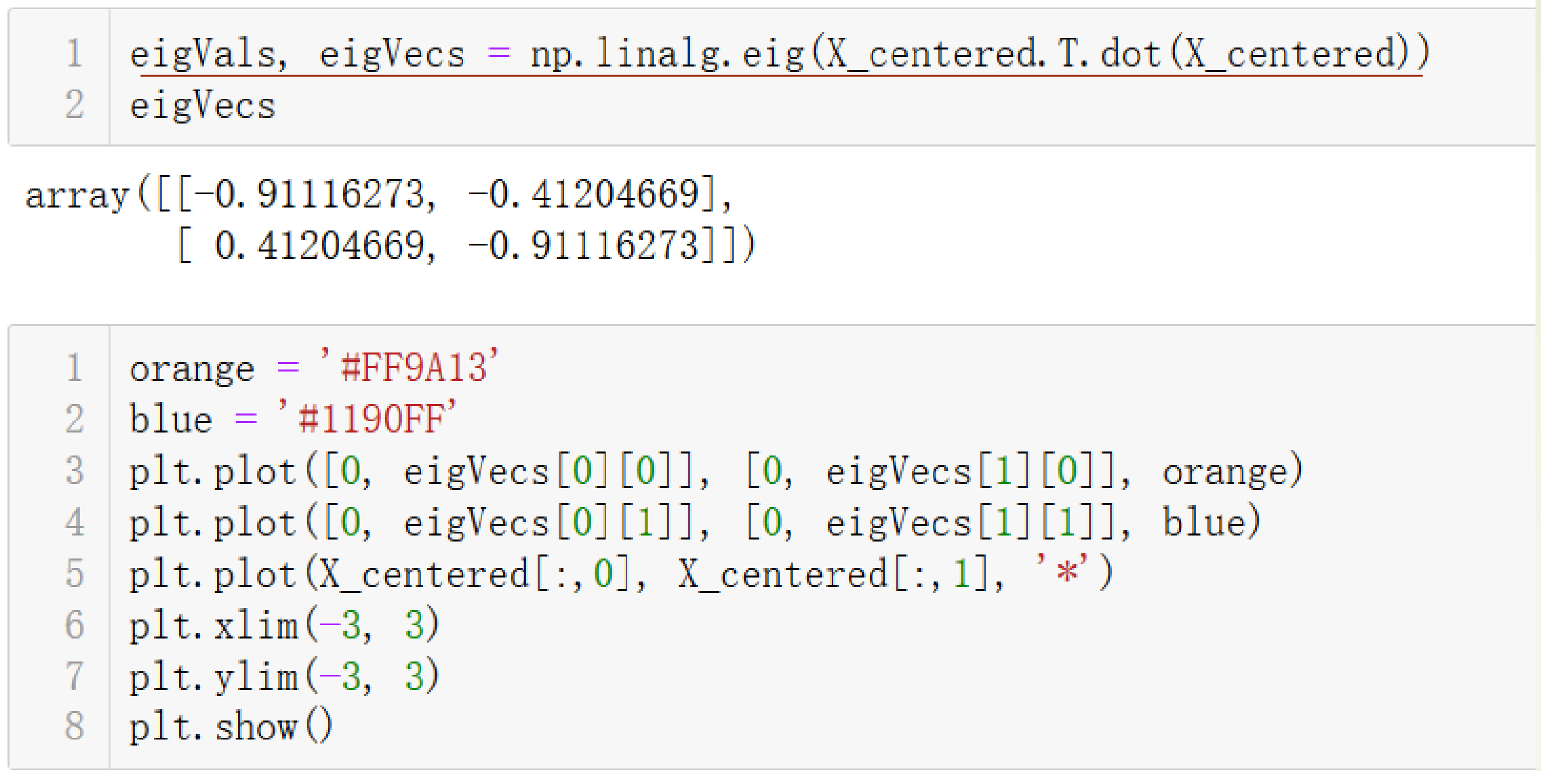


```
eigVals, eigVecs = np.linalg.eig(X_centered.T.dot(X_centered))
eigVecs
```

```
array([[-0.91116273, -0.41204669],
       [ 0.41204669, -0.91116273]])
```

```
orange = '#FF9A13'
blue = '#1190FF'
plt.plot([0, eigVecs[0][0]], [0, eigVecs[1][0]], orange)
plt.plot([0, eigVecs[0][1]], [0, eigVecs[1][1]], blue)
plt.plot(X_centered[:,0], X_centered[:,1], '*')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.show()
```

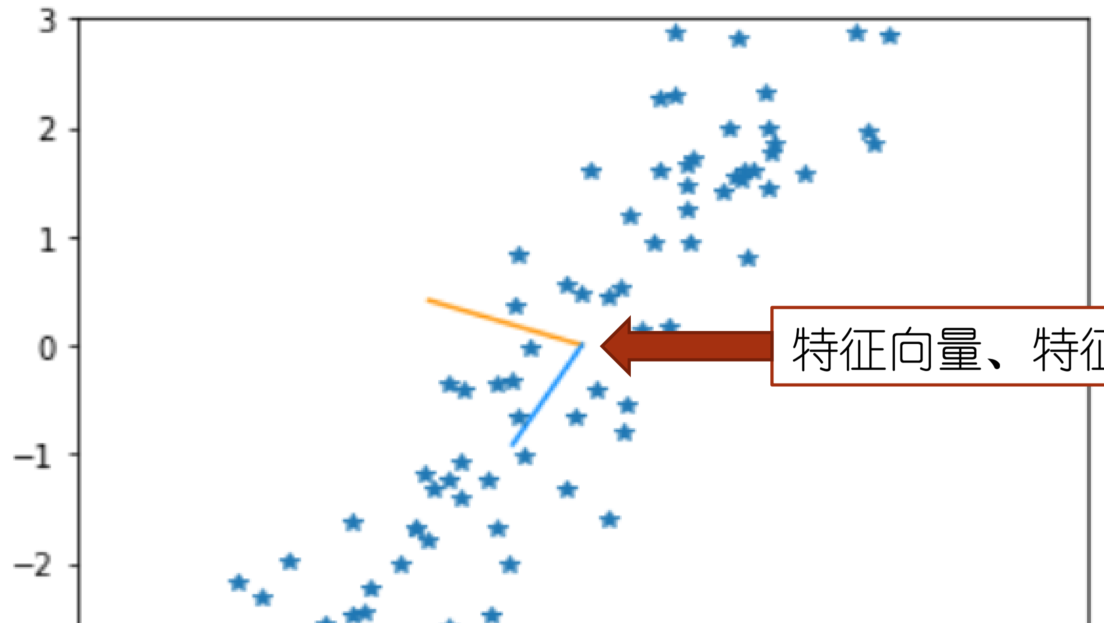




```
eigVals, eigVecs = np.linalg.eig(X_centered.T.dot(X_centered))
eigVecs
```

```
array([[-0.91116273, -0.41204669],
       [ 0.41204669, -0.91116273]])
```

```
orange = '#FF9A13'
blue = '#1190FF'
plt.plot([0, eigVecs[0][0]], [0, eigVecs[1][0]], orange)
plt.plot([0, eigVecs[0][1]], [0, eigVecs[1][1]], blue)
plt.plot(X_centered[:,0], X_centered[:,1], '*')
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.show()
```



特征向量、特征值的物理意义