

# Python与数据科学导论-06

## —— Numpy基础



胡俊峰 2022/03/17  
北京大学计算机学院

# 回顾总结一下上次课的内容

- 采用回调函数的设计方案可以把待执行的函数封装成一个‘任务’
- 任务管理队列及任务封装机制相结合可以实现异步并发框架下的协程运行方案
- `async/await`语法开放了一种更完善的异步并发协程方案 (python 3.7+)

# async/await的异步并发管理机制

- 可以声明异步对象、实现异步调用、任务队列管理、休眠与事件唤醒
- `asyncio`: 声明异步对象 (awaitable objects)
- `await`: 异步调用/挂起 命令
- `asyncio.run(main())`
- `loop.run_until_complete(tasklist)`

<https://docs.python.org/zh-cn/3/library/asyncio-task.html>

<https://zhuanlan.zhihu.com/p/27258289>

# Python编程内容总结

- 数据是对象，也可以是一个生成表达式或生成器函数
- 在函数闭包模式下，函数是被加工的数据对象，可以被动态生成并输出
- 类可以认为是对象的生成器，其本身也是对象
- 派生生成器对象、可执行对象的类可以看作是函数的母函数，实现类似闭包的功能
- 进程是程序，例程是可并发的子程序，协程是可实现动态响应的伺服函数
- 类设计模式总结了一些常用场景下的类框架模式供参考
- 事件循环方案提供了任务封装与调度的思想，为开发数据服务系统提供思路

# 本次课内容

- Numpy基础:
  - ndarray对象及基本操作
  - Numpy与矩阵运算
- 矩阵的特征值与特征向量
- 数据可视化的软件包：Matplotlib

# Numpy基础


## NumPy

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

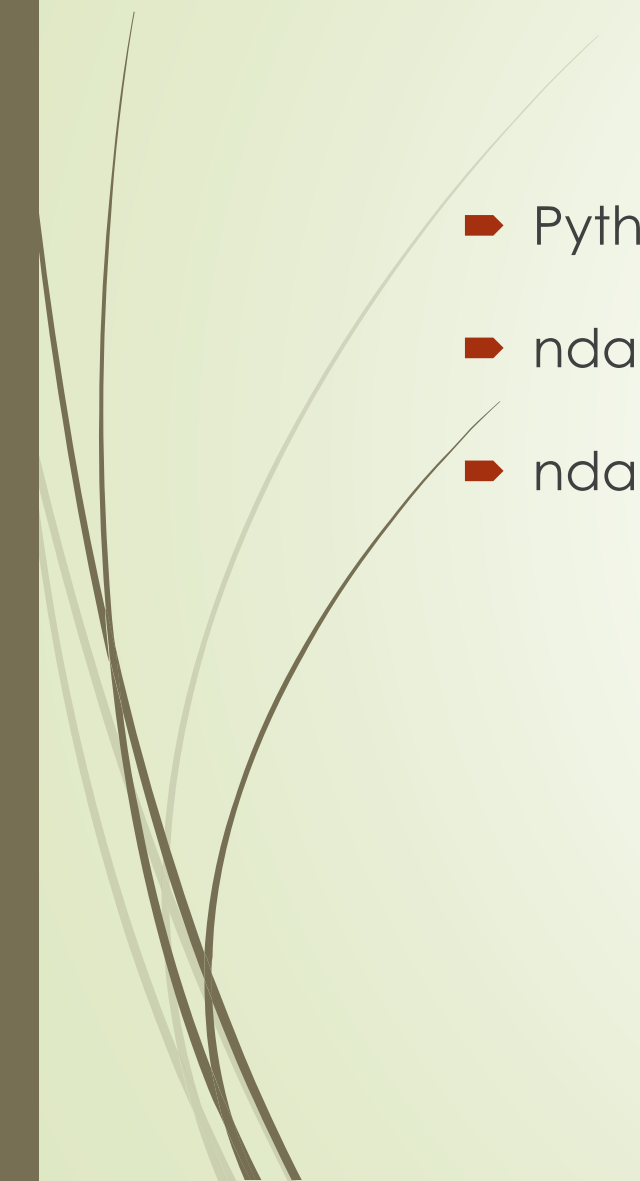
- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

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

**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

```
: 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])
```



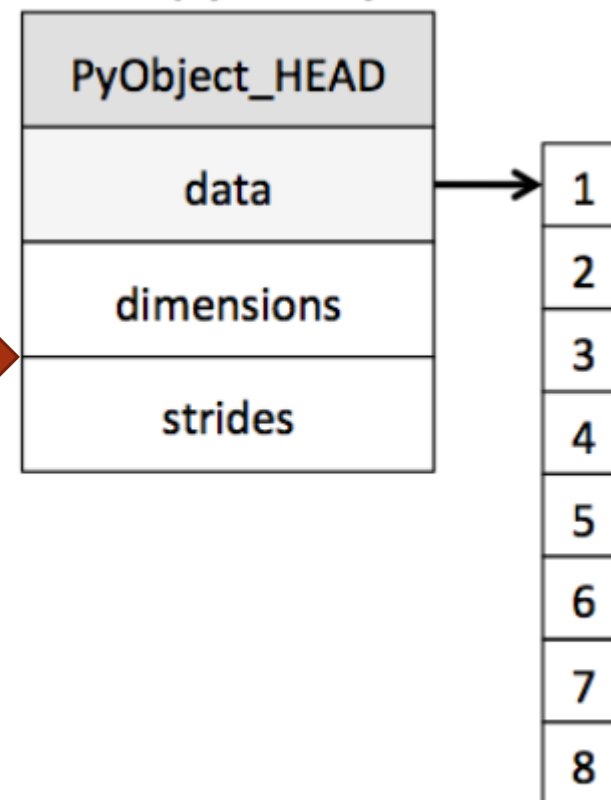
## 高效的固定类型的矩阵数据处理方案

```
1 import numpy as np
2
3 ar = np.array([3.14, 4, 2, 3])
4 print(type(ar))
5 ar
```

```
<class 'numpy.ndarray'>
```

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

Numpy Array



# 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)
```

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

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

```
[3 4]
```

```
[1 1]
```

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

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

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

## 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}
```

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

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

一些常见的初始化矩阵方法

```
1 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)
```

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

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

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

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

一些常见的初始化矩阵方法（续）

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

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

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

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

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

Stride属性可被用于加速高维数组访问

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

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

```
[[[ 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是二维数组
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)

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




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

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

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

Add, subtract, multiply, divide

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



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

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

```
1 # Elementwise square root; produces the array
2 y = np.sqrt(y)
3 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]]
```

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)
```

# 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:

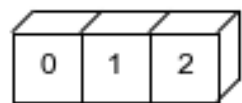
```
a = np.array([[1, 2, 3, 4], [10, 20, 30, 40]])
b = np.array([100, 100, 100, 100])
print(a + b)           # 行 对矩阵广播
```

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

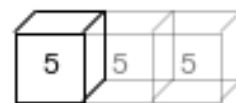
```
[[101 102 103 104]
 [110 120 130 140]]
```

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

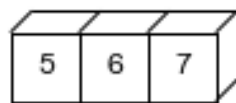
`np.arange(3)+5`



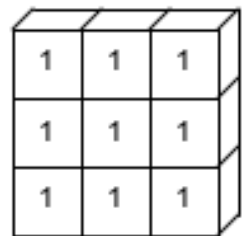
+



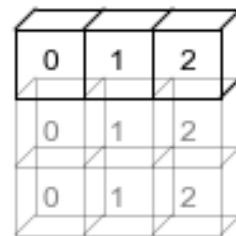
=



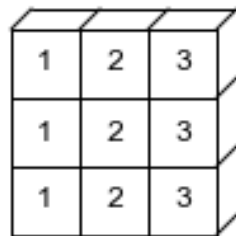
`np.ones((3, 3)) + np.arange(3)`



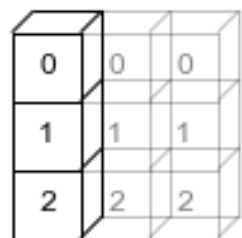
+



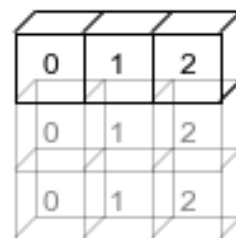
=



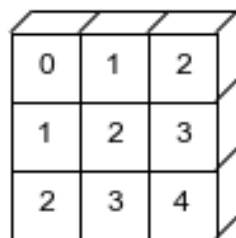
`np.arange(3).reshape((3, 1)) + np.arange(3)`



+



=





## 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换
```

```
[[0 1 2]
```

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

```
[[ 0  2  4]
```

```
 [ 6  8 10]]
```

```
[2 8]
```

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

```
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))
```

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

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

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

```
[['aaaaddddd' 'bbbbeeeee' 'ccccfffff']
 ['aaaaaaddddd' 'bbbbbeeeee' 'ccccccfffff']] (2, 2, 3)
```

## 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)
```

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

```
np.inner(a, b) # 对应字符串相乘相加
```

```
array([[['b', 'd'],
        'aabb'bb', 'ccdd' ]],

      [['aaaabb'bb'bb', 'ccccdd'dd' ],
       ['aaaaaabb'bb'bb'bb', 'ccccccdd'dd'dd' ]]], dtype=object)
```

```
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'],
        'aacc', 'bbdd' ]],

      [['aaaac'cccc', 'bbbbdd'dd' ],
       ['aaaaaac'cccc'cc', 'bbbbbbdd'dd'dd' ]]], dtype=object)
```



## `np.outer(a, b)`, 向量的外积。

Given two vectors,  $a = [a_0, a_1, \dots, a_M]$  and  $b = [b_0, b_1, \dots, b_N]$ , the outer product [1] is:

$[[a_0b_0 \ a_0b_1 \ \dots \ a_0b_N]$

$[a_1b_0 \ .$

$[ \dots \ .$

$[aMb_0 \ aMb_N ]]$

```
: c = np.ones((3,))
d = np.linspace(-1, 2, 5) ←
print(c, d)
rl = np.outer(c, d)
rl
```

$[1. \ 1. \ 1.] \ [-1. \ \underline{-0.25} \ 0.5 \ 1.25 \ 2. \ ]$

```
: array([[ -1.   , -0.25,  0.5 ,  1.25,  2.   ],
        [ -1.   , -0.25,  0.5 ,  1.25,  2.   ],
        [ -1.   , -0.25,  0.5 ,  1.25,  2.   ]])
```

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

```
1 import numpy as np
2
3 a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
4
5 # Use slicing to pull out the subarray consisting of the first 2 rows
6 # and columns 1 and 2; b is the following array of shape (2, 2):
7 # [[2 3]
8 #  [6 7]]
9 b = a[:2, 1:3]
10 print (b)
```

← : 引导的区间，左闭右开

*each dimension of the array:*

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

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

```
1 print (a[0, 1])
2 b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
3 print (a[0, 1])
4 c = b.copy()      ← 或者用take操作
5 c[0, 0] = 66
6 print (a[0, 1])
```

2

77

77

## 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。 结果
```

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

- 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], :]    copy第2行作为新矩阵的第一行
print (row_r1, row_r1.shape)
print (row_r2, row_r2.shape)
print (row_r3, row_r3.shape)
```

```
[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]
```

# 可视化软件包matplotlib

## <https://www.matplotlib.org.cn/tutorials/>



The screenshot shows the Matplotlib Chinese website. The browser address bar displays <https://www.matplotlib.org.cn/tutorials/>. The page header includes the Matplotlib logo and the text "Matplotlib 中文网". The main heading is "教程" (Tutorial). Below it, a paragraph states: "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。" (This page contains more in-depth guides on using Matplotlib. It is divided into beginner, intermediate, and advanced sections, as well as sections covering specific topics). Another paragraph follows: "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](#)。" (For shorter examples, see our [example gallery](#). You can also find [external resources](#) and [FAQs](#) in our [user guide](#).) The section "序言" (Preface) begins with the text: "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。" (These tutorials cover the basic knowledge of creating visualizations using Matplotlib, as well as some best practices for effectively using the software package.) Below the text are six thumbnail images representing different tutorial topics: 1. "Simple Plot" showing linear, quadratic, and cubic functions. 2. "Usage Guide" showing a bar chart titled "Company Revenue". 3. "Pyplot tutorial" showing a simple line plot. 4. "Sample plots in Matplotlib" showing a 2x2 grid of different plot types: histogram, scatter, line, and heatmap. 5. "Image tutorial" showing a grayscale image of a beetle. 6. A line plot showing a time series or signal.

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。

Simple Plot

Usage Guide

Pyplot tutorial

Sample plots in Matplotlib

Image tutorial

## 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:

```
1 import matplotlib as mpl
2 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:

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

## Plotting from a script

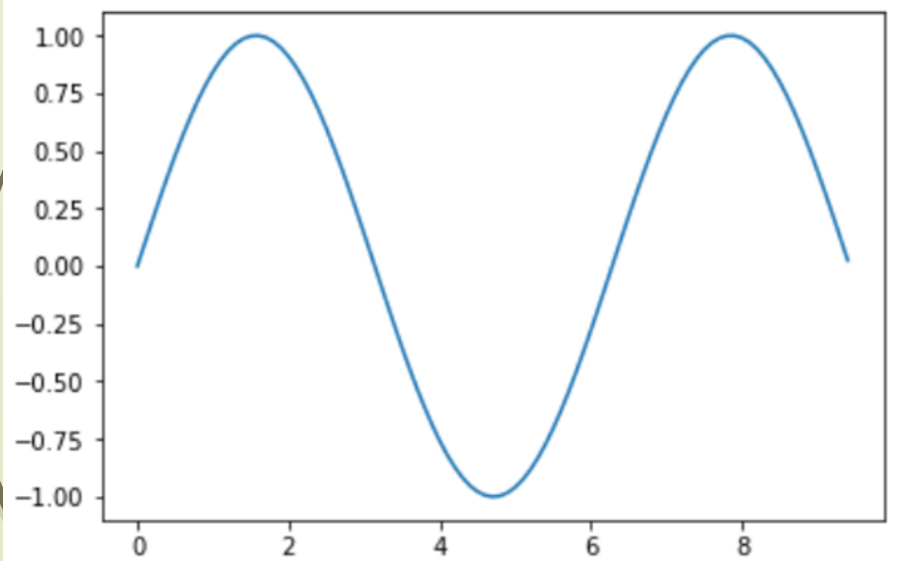
If you are using Matplotlib from within a script, the function `plt.show()` is your friend. `plt.show()` starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.

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()
```

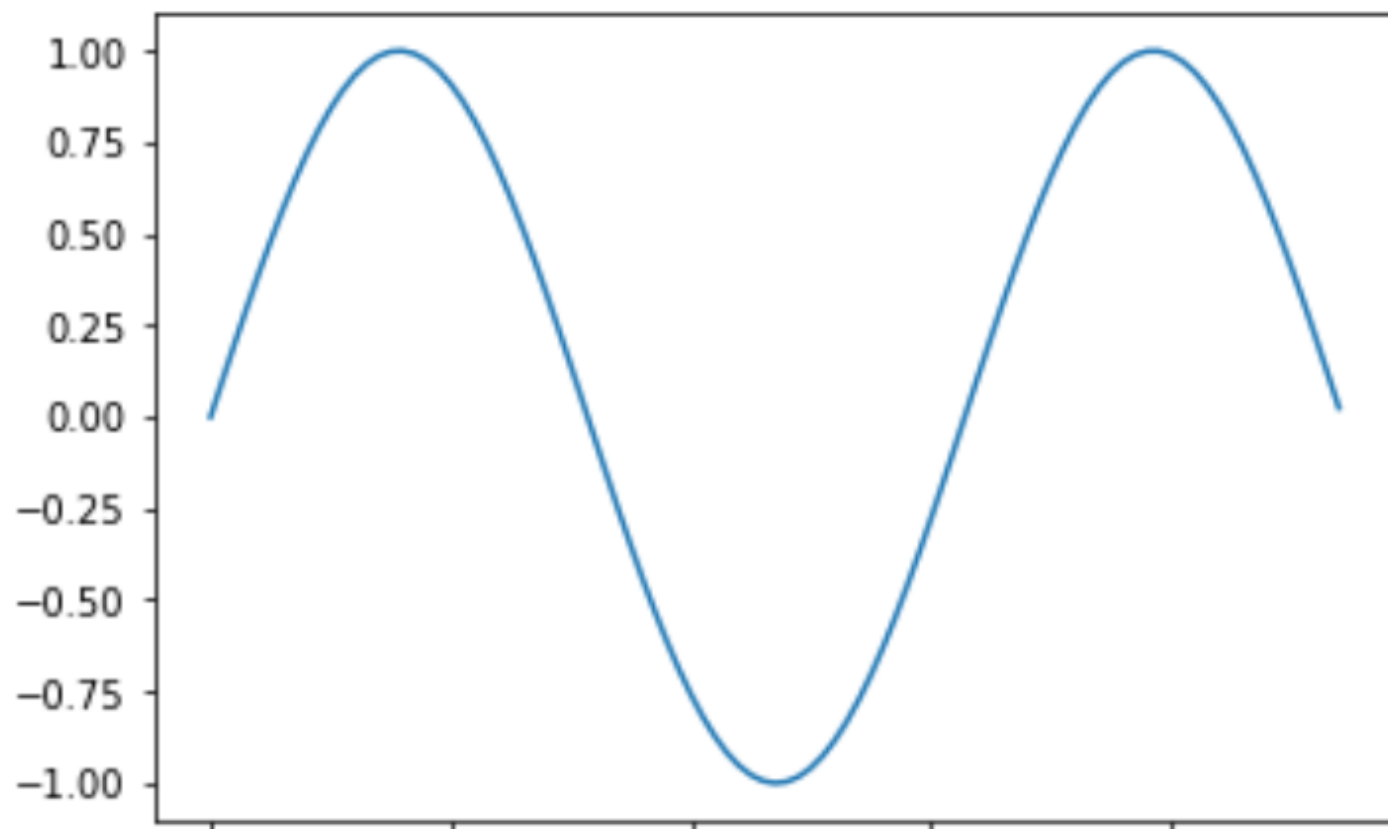
# Plotting —— 二维画图软件包

```
1 import matplotlib.pyplot as plt
2 x = np.arange(0, 3 * np.pi, 0.1) #arange函数用于包
3 y = np.sin(x)
4
5 plt.plot(x, y) # 按坐标画图
```



```
1 import matplotlib.pyplot as plt
2 x = np.arange(0, 3 * np.pi, 0.1) #arange函数用于创建等差数组
3 y = np.sin(x)
4
5 plt.plot(x, y) # 按坐标画图
```

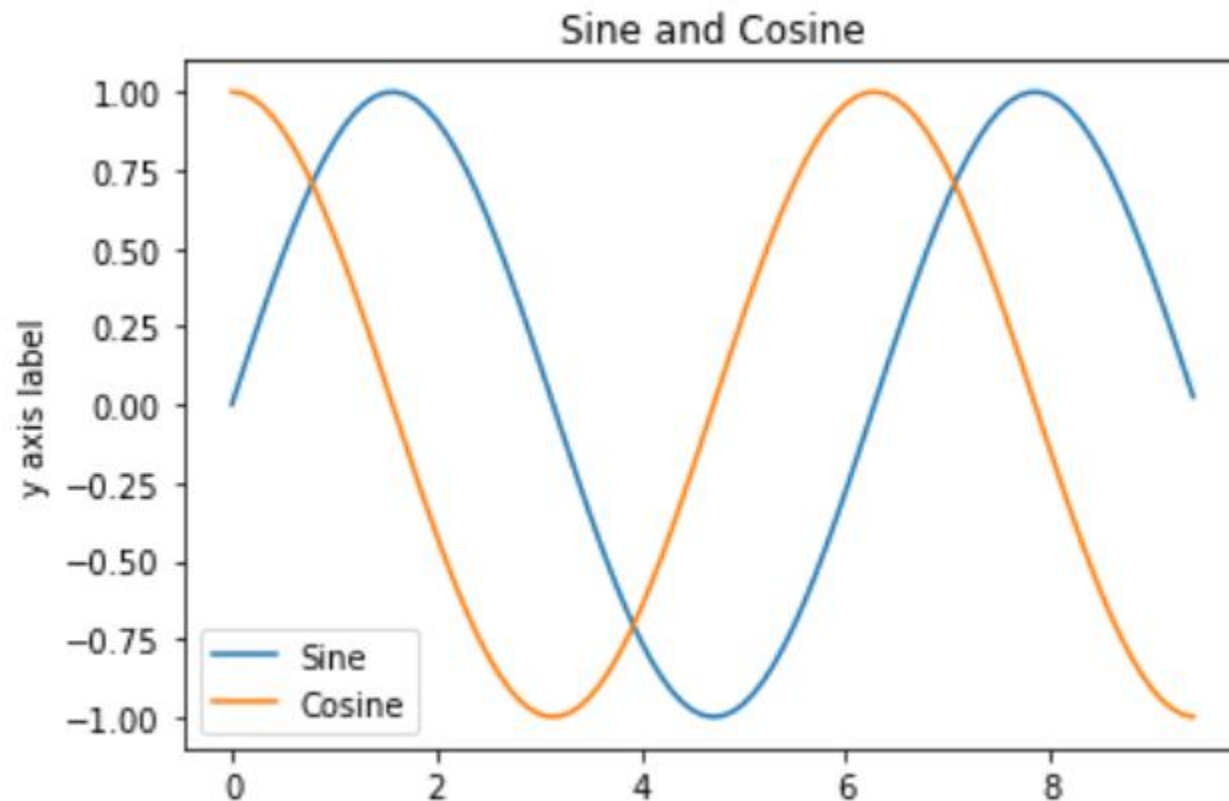
[<matplotlib.lines.Line2D at 0x1b20b857128>]



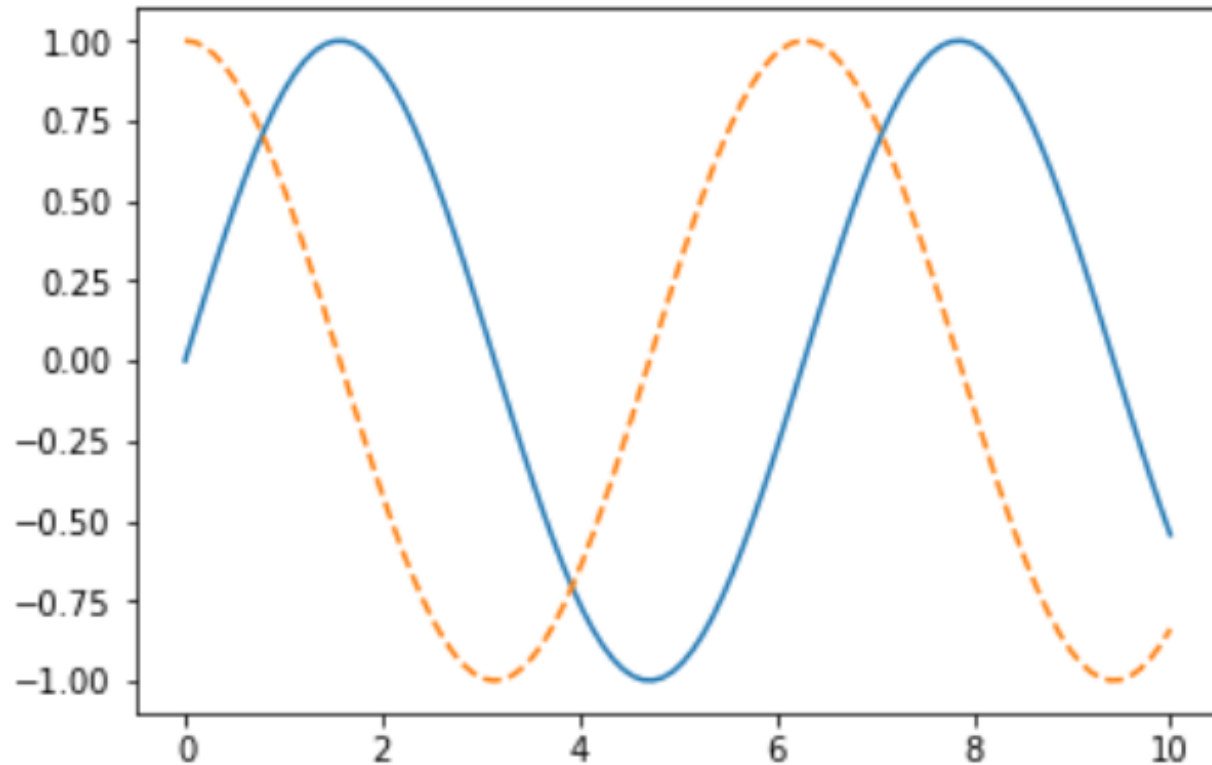


```
1 y_sin = np.sin(x)
2 y_cos = np.cos(x)
3
4 plt.plot(x, y_sin)
5 plt.plot(x, y_cos)
6 plt.xlabel('x axis label')
7 plt.ylabel('y axis label')
8 plt.title('Sine and Cosine')
9 plt.legend(['Sine', 'Cosine'])
```

<matplotlib.legend.Legend at 0x1d7e537b0b8>



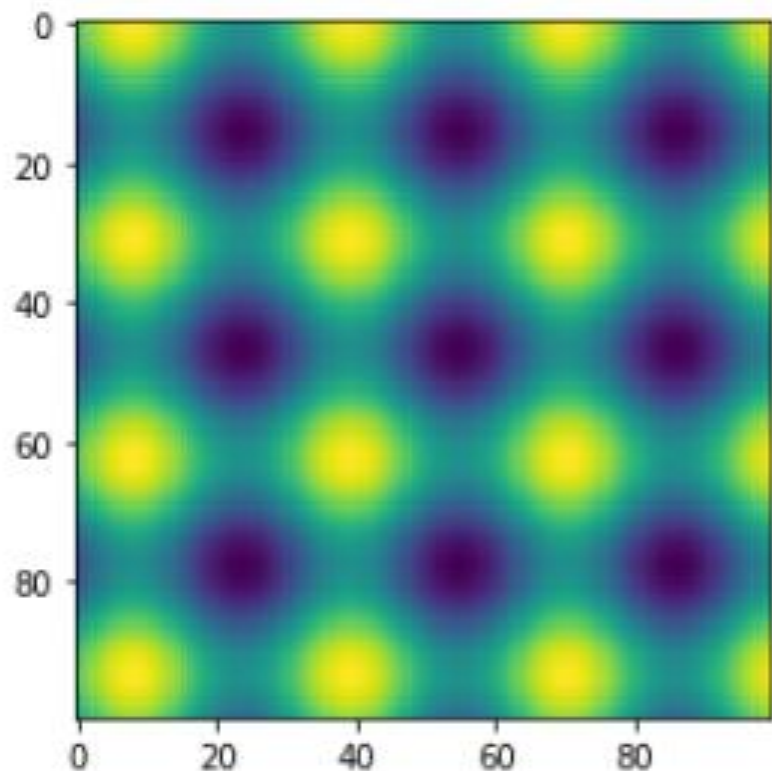
```
1 import numpy as np
2 import matplotlib as mpl
3 import matplotlib.pyplot as plt
4 x = np.linspace(0, 10, 100)
5
6 #fig = plt.figure()
7 plt.plot(x, np.sin(x), '-')
8 plt.plot(x, np.cos(x), '--');
```





```
: # 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:

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

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

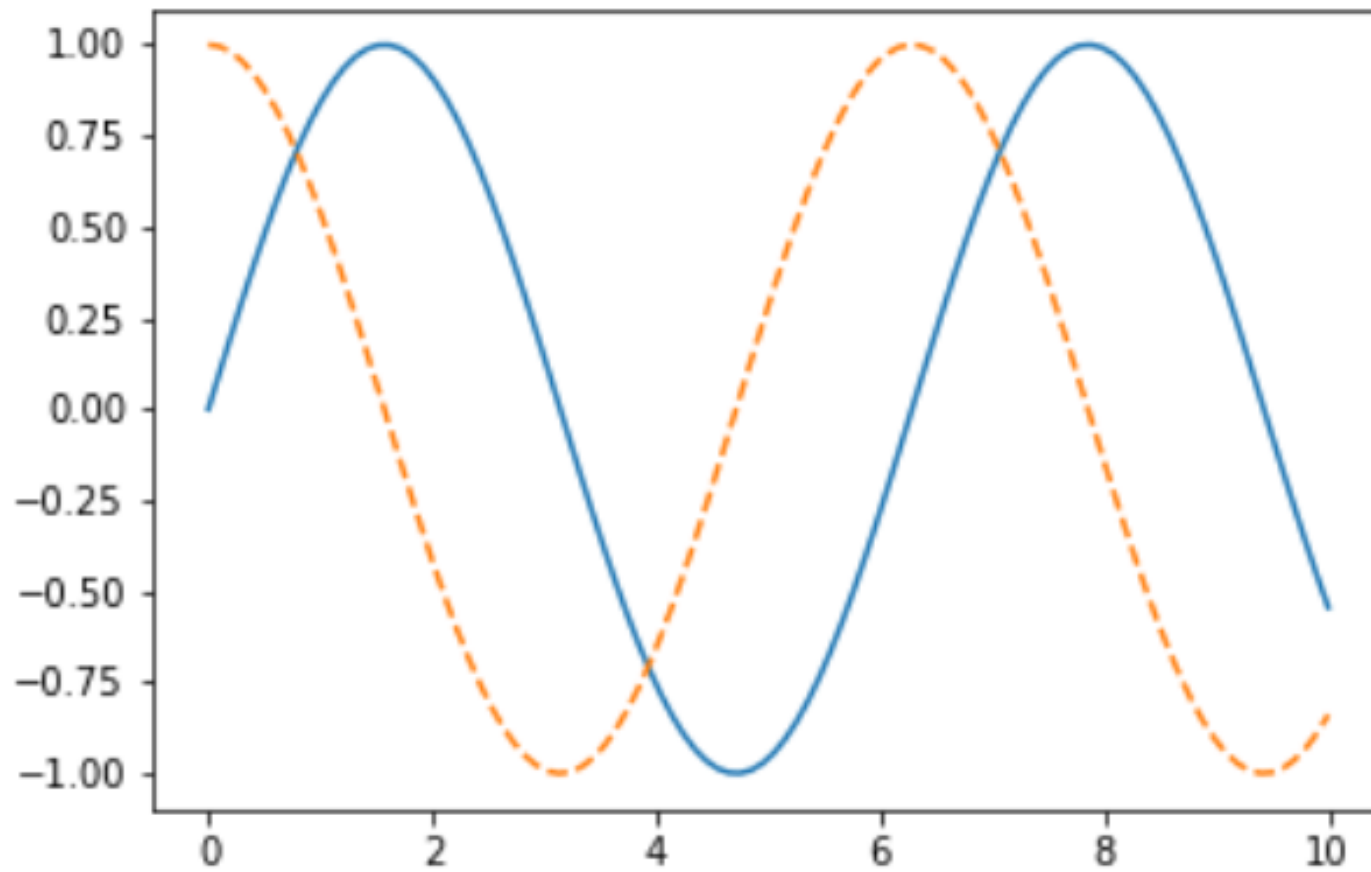
```
1 !dir my_figure.png
```

驱动器 C 中的卷是 Windows  
卷的序列号是 368A-7FED

C:\Users\hjfp\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

```
1 from IPython.display import Image
2 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*)

[\[source\]](#)

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  $ax = b$ .

**Parameters:** *a* : ( $\dots, M, M$ ) *array\_like*

Coefficient matrix.

*b* :  $\{(\dots, M,), (\dots, M, K)\}$ , *array\_like*

Ordinate or “dependent variable” values.

**Returns:** *x* :  $\{(\dots, M,), (\dots, M, K)\}$  *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。

```
1 def func1():
2     x = np.linalg.inv(B) @ C
3
4 def func2():
5     x = np.linalg.solve(B, C)
6
7
8 t = timeit('func1()', 'from __main__ import func1', number=100)
9 print('func1=', t)
10
11 t = timeit('func2()', 'from __main__ import func2', number=100)
12 print('func2=', t)
13
14
```

func1= 0.8877446270053042

func2= 0.29727534799894784

# 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

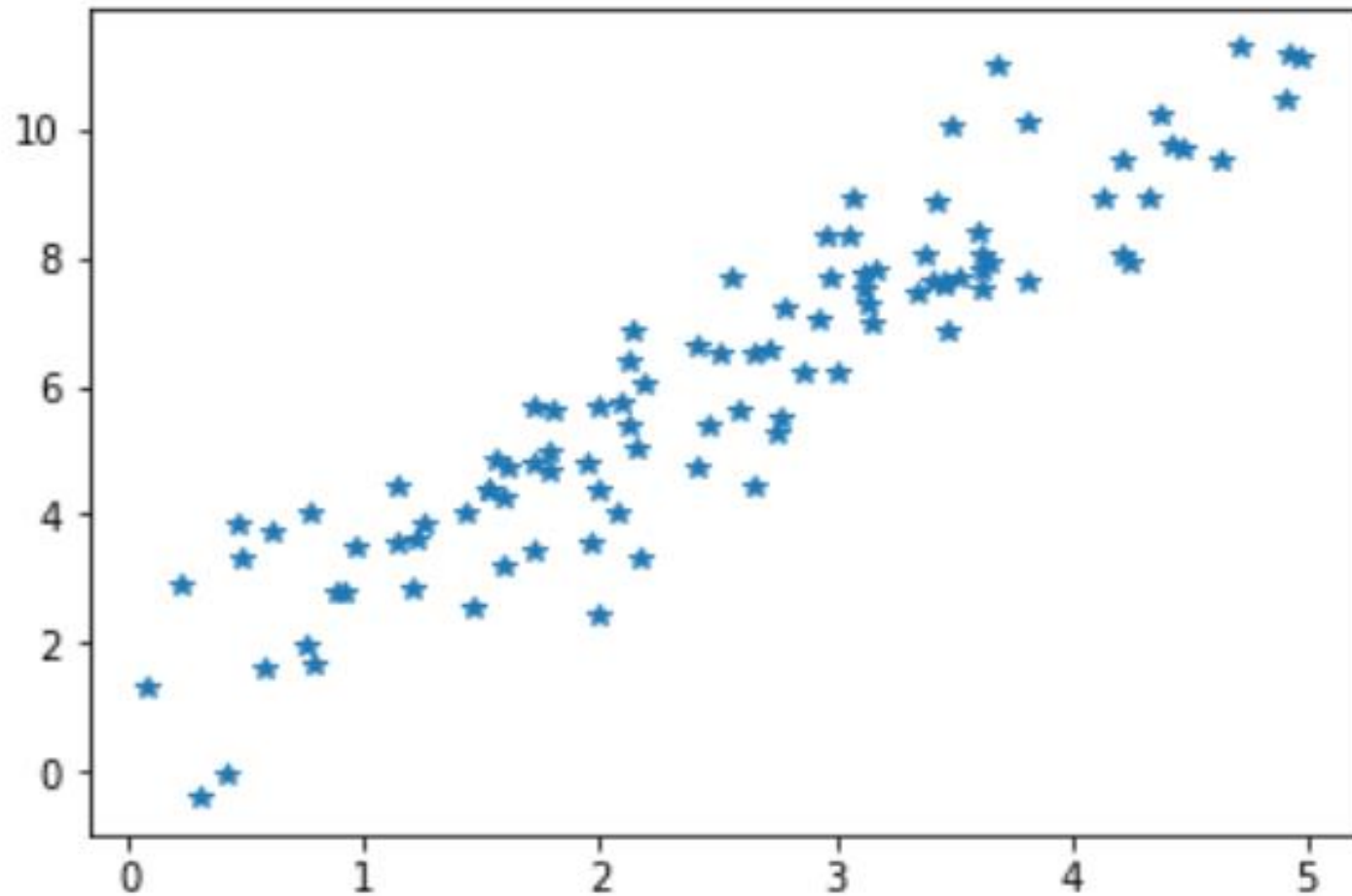
## 生成二维数据：

```
1 np.random.seed(123)
2 x = 5*np.random.rand(100)
3 y = 2*x + 1 + np.random.randn(100)
4
5 x = x.reshape(100, 1)
6 y = y.reshape(100, 1)
7
8 X = np.hstack([x, y]) # 水平堆叠
9 X.shape
```

(100, 2)

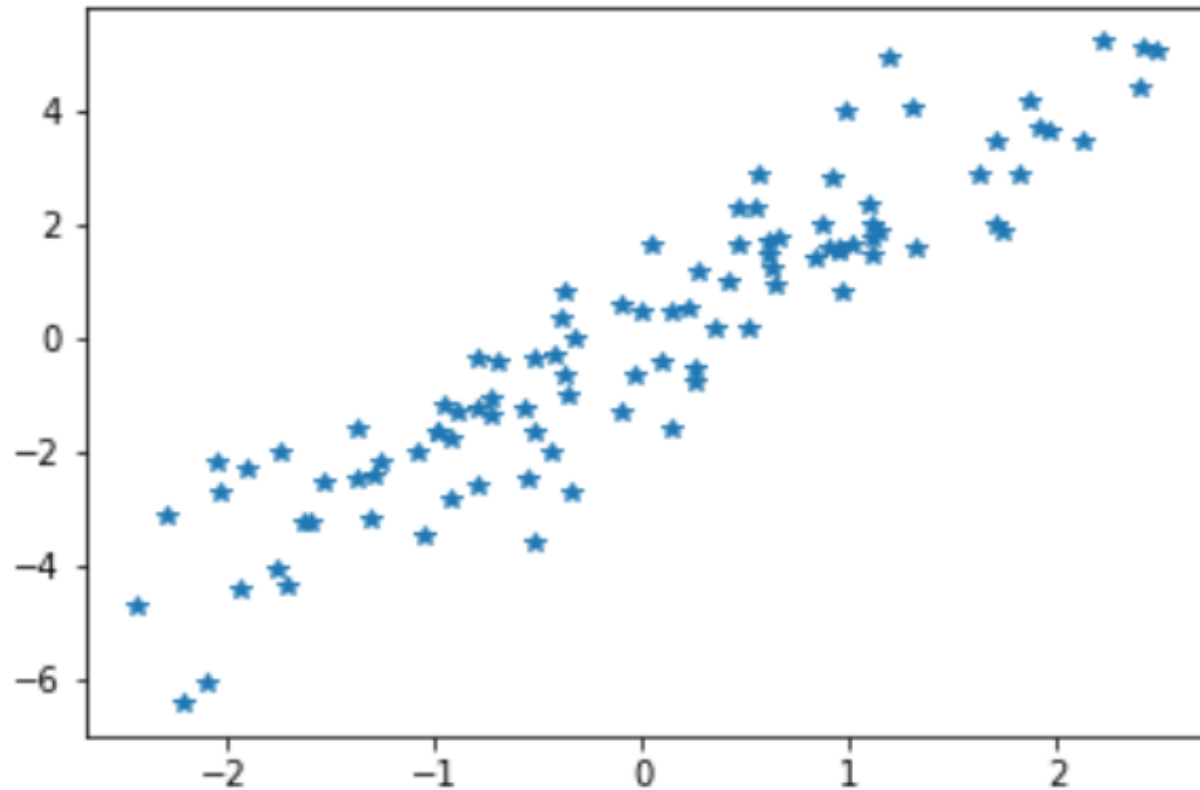


```
1 %matplotlib inline
2 from matplotlib import pyplot as plt
3 plt.plot(X[:,0], X[:,1], '*')
4 plt.show()
```





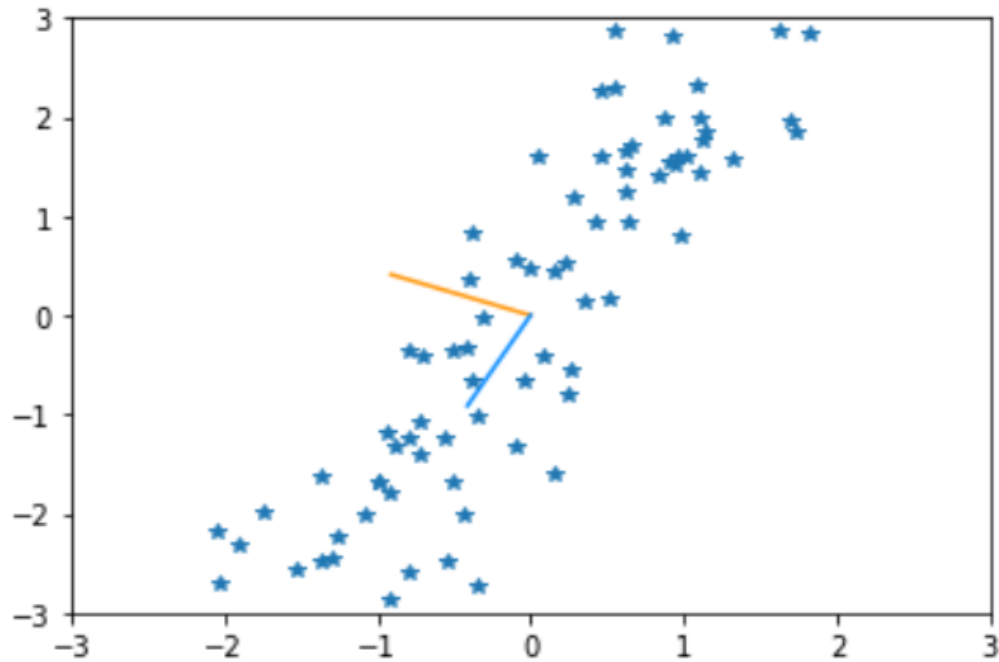
```
1 def centerData(X):  
2     X = X.copy()  
3     X -= np.mean(X, axis = 0)  
4     return X  
5  
6 X_centered = centerData(X)  
7 plt.plot(X_centered[:,0], X_centered[:,1], '*')  
8 plt.show()
```



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

```
array([[ -0.9116273, -0.41204669],
       [ 0.41204669, -0.9116273]])
```

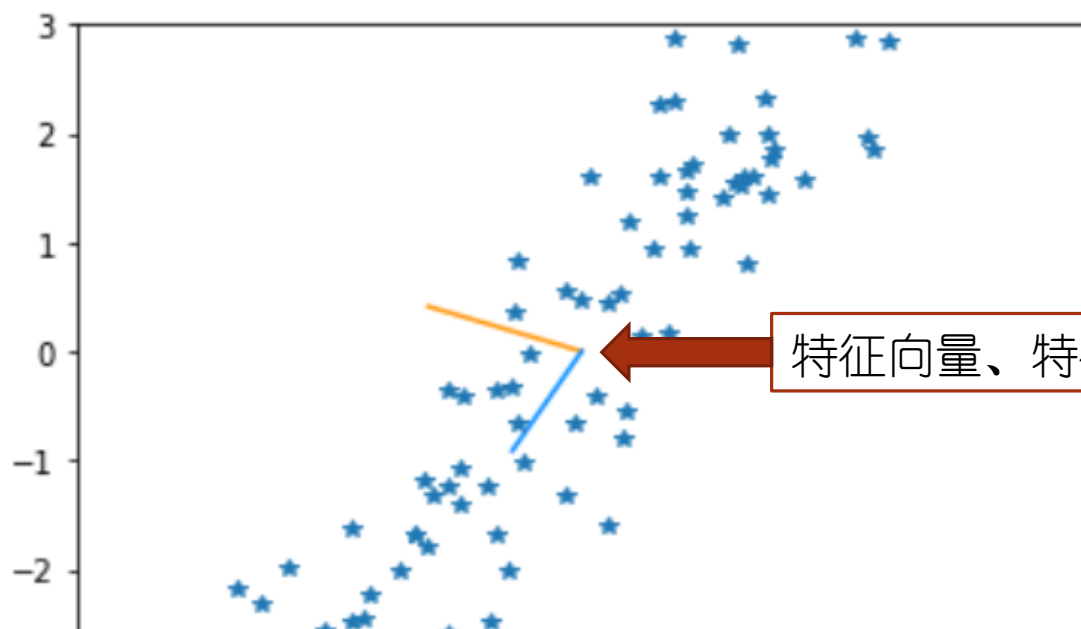
```
1 orange = '#FF9A13'
2 blue = '#1190FF'
3 plt.plot([0, eigVecs[0][0]], [0, eigVecs[1][0]], orange)
4 plt.plot([0, eigVecs[0][1]], [0, eigVecs[1][1]], blue)
5 plt.plot(X_centered[:,0], X_centered[:,1], '*')
6 plt.xlim(-3, 3)
7 plt.ylim(-3, 3)
8 plt.show()
```



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

```
array([[ -0.9116273, -0.41204669],
       [ 0.41204669, -0.9116273]])
```

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



特征向量、特征值的物理意义

# 主成分分析 (PCA) 与特征空间降维

- 概率与信息熵
- 主成分分析PCA与特征工程
  - 特征的信息量与区分度
  - 特征正交化与PCA降维
- 基于（标准）正交空间的聚类-分类算法

## 回顾两个数学概念：概率 - 信息量

- ➡ 概率：  $P_i = F_i / \sum_i F_i$  （古典概型，也称频率模型）
- ➡ 信息量：  $H_i = -\log P_i$

概率越小，信息量越大，概率越大，信息量越小

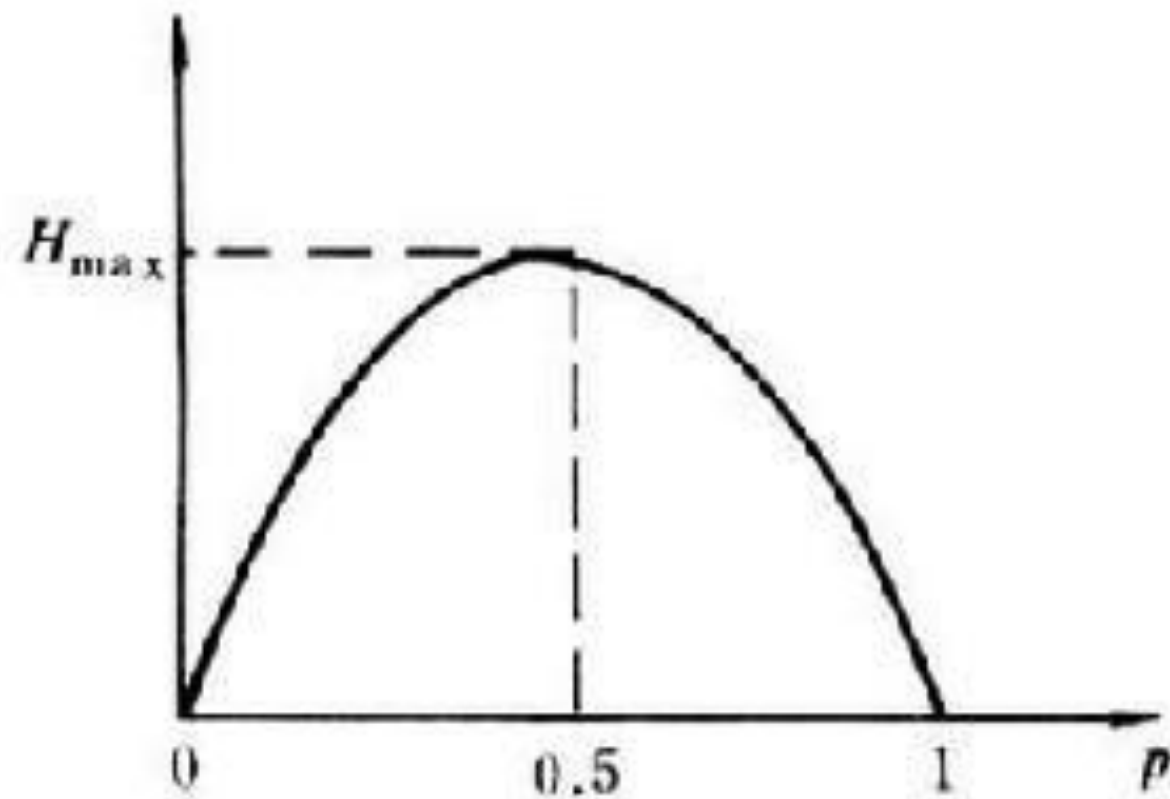
再进一步：编码系统的信息量（信息熵）

$$H(U) = E[-\log p_i] = - \sum_{i=1}^n p_i \log p_i$$

2元编码系统均匀分布的信息熵：

$$H_2 = 2 * (-1/2 \log(1/2)) = 1 \text{ bit}$$

4元编码系统均匀分布的信息熵  
= ?

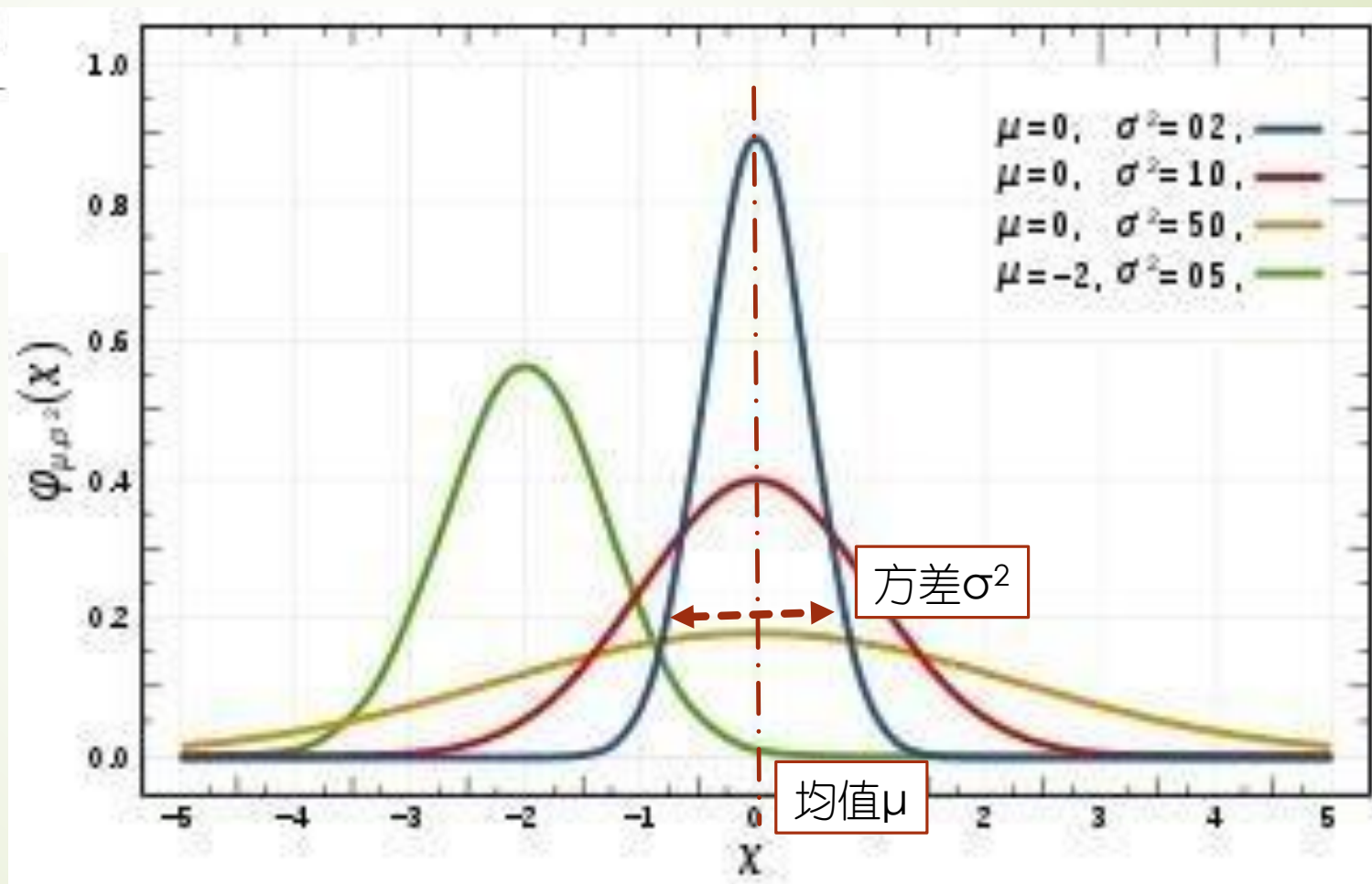


二元信源的熵函数

# 概率分布 - 方差 - 特征区分度

$$p(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

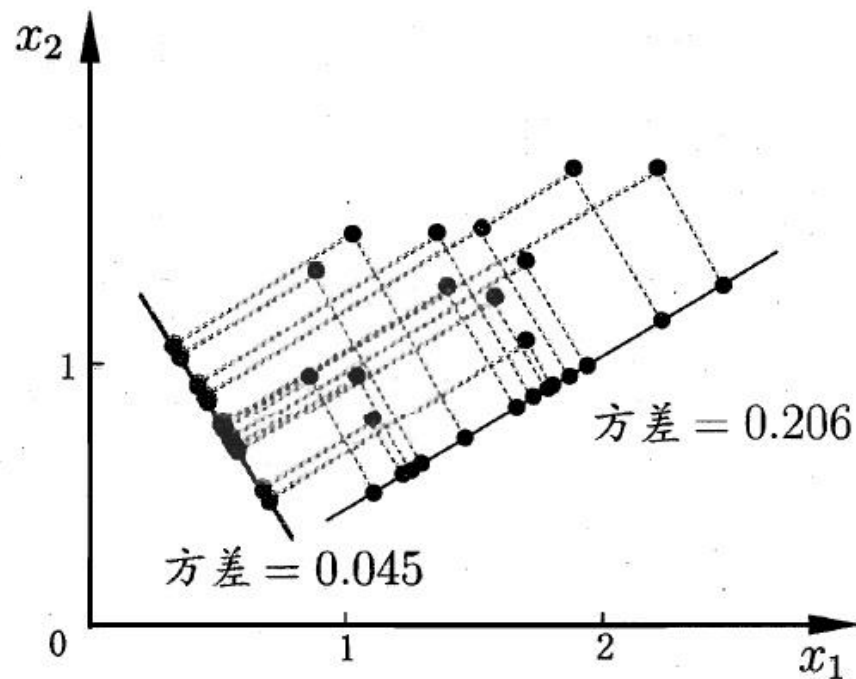
一个特征分布的方差越大，  
其信息量即区分度也就越高





# 特征空间、特征分布的方差、特征空间变换

- 假定样本的所有属性构成了N维的向量空间，每个样本都是空间中的一个向量
- 转过来看，每个属性在样本集中的取值，则构成了该属性在样本集中的一个分布
- 一个‘好’的属性，意味着在样本空间中的分布有大的方差（区分度）
- 特征空间变换的目标是使得前K维特征相互正交且具备最大的区分度（方差）





# 样本集下的特征的协方差矩阵

- $C_{\vec{X}}(i; j) = Cov_{X_i, X_j} = E \left( (X_i - E(X_i)) (X_j - E(X_j)) \right)$
- 对于样本特征  $\vec{X}^{(1)}, \dots, \vec{X}^{(N)}$ , 需要先让  $\vec{X}^{(i)} \leftarrow \vec{X}^{(i)} - \frac{1}{N} \sum_{i=1}^N \vec{X}^{(i)}$  (属性值中心化)

- 记样本矩阵为  $X = \begin{pmatrix} -\vec{X}^{(1)} - \\ -\vec{X}^{(2)} - \\ \dots \\ -\vec{X}^{(N)} - \end{pmatrix}$

- 可以估计协方差矩阵如下: (k是样本数)

$$C(p; q) = \frac{1}{N} \sum_{k=1}^N X_p^{(k)} X_q^{(k)} \quad (\text{向量两两相乘}) \quad \leftarrow \text{属性之间的相关性描述}$$

则:  $C = X^T X / N$

# 协方差矩阵的物理意义

$$\rightarrow C(p; q) = \frac{1}{N} \sum_{k=1}^N X_p^{(k)} X_q^{(k)}$$

对角线 $(p; p)$ 上的元素：第 $p$ 维特征的方差（偏离均值的趋势平方）

矩阵 $(p; q)$ 元的大小反映了所有样本第 $p$ 维和第 $q$ 维数据的相关性（若不相关，则为0）

# PCA（主成分分解）

—— 对于实对称矩阵，存在一组正交变换使其对角化

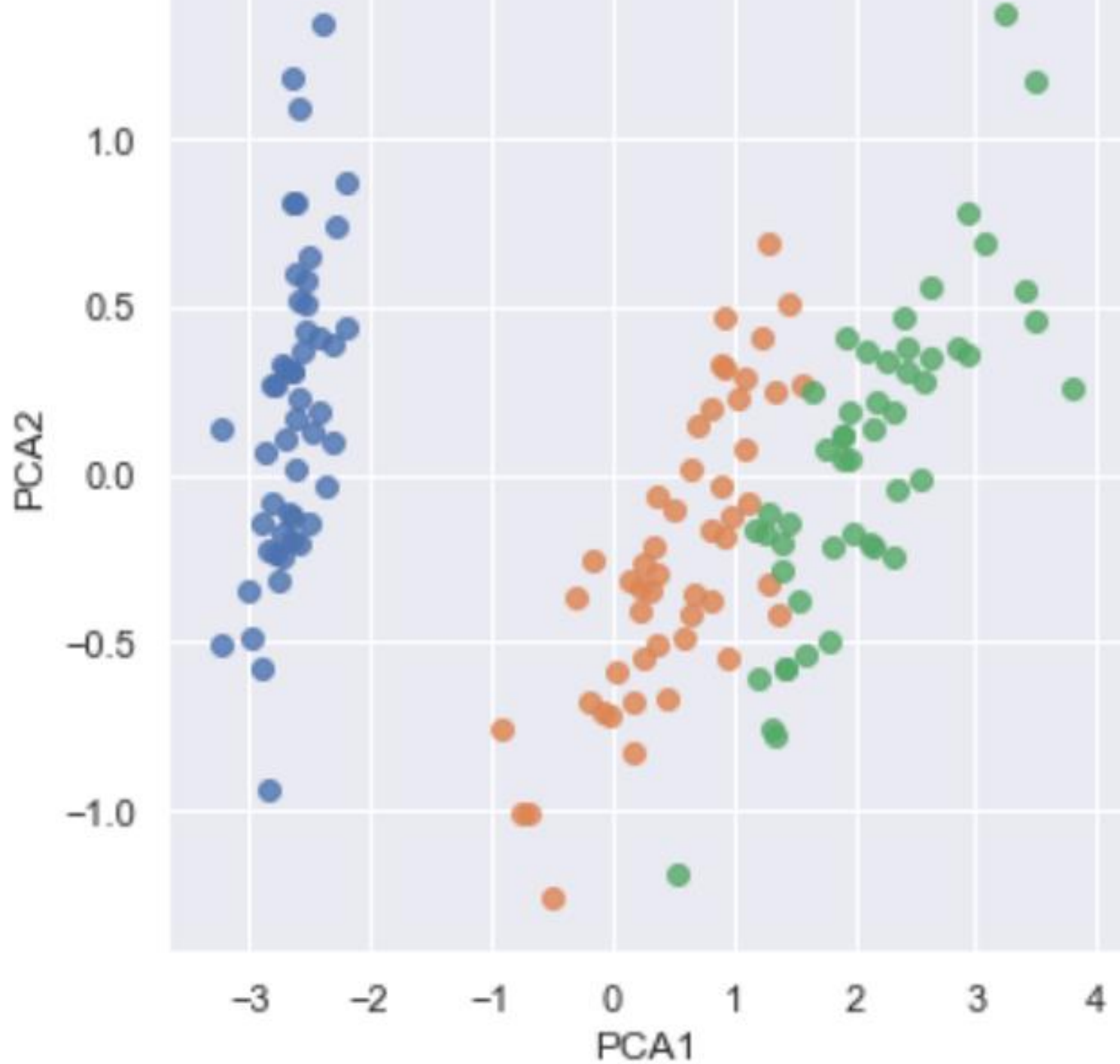
$$A = Q\Sigma Q^T = Q \begin{bmatrix} \lambda_1 & \dots & \dots & \dots \\ \dots & \lambda_2 & \dots & \dots \\ \dots & \dots & \ddots & \dots \\ \dots & \dots & \dots & \lambda_m \end{bmatrix} Q^T$$

新特征的方差

新特征空间的基底

## 数据特征与数据维度

	sepal_length	sepal_width	petal_length	petal_width	species
<b>0</b>	5.1	3.5	1.4	0.2	setosa
<b>1</b>	4.9	3.0	1.4	0.2	setosa
<b>50</b>	7.0	3.2	4.7	1.4	versicolor
<b>51</b>	6.4	3.2	4.5	1.5	versicolor
<b>100</b>	6.3	3.3	6.0	2.5	virginica
<b>101</b>	5.8	2.7	5.1	1.9	virginica



降维正交化后的特征空间  
为样本提供了更好的分布  
(需要投影到新空间)

# Ndarray的迭代器接口：



User Guide **API reference** Development



Search the docs ...

## Array objects

The N-dimensional array ( **ndarray** )

Scalars

Data type objects ( **dtype** )

Indexing

**Iterating Over Arrays**

Standard array subclasses

Masked arrays

The Array Interface

## Iterating Over Arrays

### Note

Arrays support the iterator protocol and can be iterated over like Python lists. See the [Indexing, Slicing and Iterating](#) section in the Quickstart guide for basic usage and examples. The remainder of this document presents the **nditer** object and covers more advanced usage.

The iterator object **nditer**, introduced in NumPy 1.6, provides many flexible ways to visit all the elements of one or more arrays in a systematic fashion. This page introduces some basic ways to use the object for computations on arrays in Python, then concludes with how one can accelerate the inner loop in Cython. Since the Python exposure of **nditer** is a relatively straightforward mapping of the C array iterator API, these ideas will also provide help working with array iteration from C or C++.

## Numpy数组可以返回一个（单维）迭代器

```
a = np.arange(0, 12)
a = a.reshape(3, 4)
b = (x for x in np.nditer(a)) # numpy数组的迭代器函数
print(b)
print(next(b), next(b))      # 可以返回一个生成器表达式

c = a[1:, :]
d = np.array([x for x in np.nditer(c)]) # 也可以直接生成数组
d

<generator object <genexpr> at 0x000001D4DB8D7348>
0 1

array([ 4,  5,  6,  7,  8,  9, 10, 11])
```

## nditer用于循环结构

```
a = np.arange(0, 4)
a = a.reshape(2, 2)

# for x in np.nditer(a):           # 缺省情况iter是只读

for x in np.nditer(a, op_flags = ['readwrite']): #readwrite
    x[...] = x * 10;    # 这里生成副本

print(a)

[[ 0 10]
 [20 30]]
```



## 采用nditer的multi\_indexing实现广播计算

```
it = np.nditer([a,b], flags=['multi_index']) # 广播 + multi-indexing
while not it.finished:
    ab[ it.multi_index] = it[0] + it[1] # multi-indexing的作用
    is_not_finished = it.iternext()

ab
array([[0, 2, 4],
       [3, 5, 7]])
```

