数据分析 2

NumPy： 数组和矢量计算

NumPy之于数值计算特别重要的原因之一，是因为它可以高效处理大数组的数据。

- NumPy是在一个连续的内存块中存储数据，独立于其他Python内置对象。NumPy的C语言编写的算法库可以操作内存，而不必进行类型检查或其它前期工作。比起Python的内置序列，NumPy数组使用的内存更少。

- NumPy可以在整个数组上执行复杂的计算，而不需要Python的for循环。

  性能对比
  基于NumPy的算法要比纯Python快10到100倍（甚至更快），并且使用的内存更少。

```
In [7]: import numpy as np

In [8]: %timeit my_arr = np.arange(1000000)

In [9]: %timeit my_list = list(range(1000000))
```

NumPy的ndarray：一种多维数组对象

NumPy最重要的一个特点就是其N维数组对象（即ndarray）， 该对象是一个快速而灵活的大数据集容器。你可以利用这种数组对整块数据执行一些数学运算，其语法跟标量元素之间的运算一样。

```
In [12]: import numpy as np

In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

```
In [15]: data * 10
Out[15]:
array([[ -2.0471,   4.7894,  -5.1944],
       [ -5.5573,  19.6578,  13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

ndarray是一个通用的同构数据多维容器，所有元素必须是相同类型的

```
# 取维度大小
data.shape
# 取数据数据类型
data.dtype
```

创建ndarray

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

嵌套序列（比如由一组等长列表组成的列表）将会被转换为一个多维数组

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
```

```
       [5, 6, 7, 8]])

# 取维度
arr2.ndim
arr2.shape
arr2.dtype
```

zeros和ones分别可以创建指定长度或形状的全0或全1数组。empty可以创建一个没有任何具体值的数组

```
In [29]: np.zeros(10)
Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [31]: np.empty((2, 3, 2))
np.ones(10)
```

np.empty返回的都是一些未初始化的垃圾值

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

函数	说明
array	将输入数据（列表、元组、数组或其它序列类型）转换为 ndarray。要么推断出 dtype，要么特别指定 dtype。默认直接复制输入数据
asarray	将输入转换为 ndarray，如果输入本身就是一个 ndarray 就不进行复制
arange	类似于内置的 range，但返回的是一个 ndarray 而不是列表
ones,ones_like	根据指定的形状和 dtype 创建一个全 1 数组。one_like 以另一个数组为参数，并根据其形状和 dtype 创建一个全 1 数组
zeros,zeros_like	类似于 ones 和 ones_like，只不过产生的是全 0 数组而已
empty,empty_like	创建新数组，只分配内存空间但不填充任何值
full,full_like	用 fill value 中的所有值，根据指定的形状和 dtype 创建一个数组。full_like 使用另一个数组，用相同的形状和 dtype 创建
eye,identity	创建一个正方的 N×N 单位矩阵（对角线为 1，其余为 0）

ndarray的数据类型

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

astype方法明确地将一个数组从一个dtype转换成另一个dtype

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

将浮点数转换成整数，则小数部分将会被截取删除

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
Out[42]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

调用astype总会创建一个新的数组（一个数据的备份）

NumPy数组的运算

不用编写循环即可对数据执行批量运算。NumPy用户称其为矢量化（vectorization）。大小相等的数组之间的任何算术运算都会将运算应用到元素级

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

数组与标量的算术运算会将标量值传播到各个元素

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr * 0.5
```

大小相同的数组之间的比较会生成布尔值数组

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[  0.,   4.,   1.],
       [  7.,   2.,  12.]])

In [59]: arr2 > arr
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

基本的索引和切片

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])
```

```
In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])

In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
9])
```

切片[:]会给数组中的所有值赋值

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

ndarray切片的一份副本而非视图，就需要明确地进行复制操作，例如arr[5:8].copy()

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])

# 两种方式一样
In [74]: arr2d[0][2]
Out[74]: 3
```

```
In [75]: arr2d[0, 2]
Out[75]: 3
```

多维数组中，如果省略了后面的索引，则返回对象会是一个维度低一点的ndarray

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
arr3d.ndim

In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
arr3d[0].ndim
```

标量值和数组都可以被赋值给arr3d[0]

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values
```

```
In [83]: arr3d
Out[83]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

切片索引

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
       [4, 5, 6]])
```

一次传入多个切片

```
In [92]: arr2d[:2, 1:]
Out[92]:
array([[2, 3],
       [5, 6]])
```

将整数索引和切片混合
选取第二行的前两列

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

选择第三列的前两行

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

“只有冒号”表示选取整个轴

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```

```
In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

布尔型索引

假设我们有一个用于存储数据的数组以及一个存储姓名的数组（含有重复项）

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

In [100]: names
```

```
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)

In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])

In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

要选择除"bob"以外的其他值，既可以使用不等于符号（!=），也可以通过~对条件进行否定

```
In [106]: names != 'Bob'
Out[106]: array([False,  True,  True, False,  True,  True,  True], dtype=bool)
```

```
In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0.    ,  0.    ],
       [ 1.669 ,  0.    ,  0.    ,  0.477 ],
       [ 3.2489,  0.    ,  0.    ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

```
In [115]: data[names != 'Joe'] = 7
```

```
In [116]: data
Out[116]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

花式索引

花式索引（Fancy indexing）是一个NumPy术语，它指的是利用整数数组进行索引

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

为了以特定顺序选取行子集，只需传入一个用于指定顺序的整数列表或ndarray即可

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
```

```
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

使用负数索引将会从末尾开始选取行

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

一次传入多个索引数组会有一点特别。它返回的是一个一维数组，其中的元素对应各个索引元组

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

# 最终选出的是元素(1,0)、(5,3)、(7,1)和(2,2)
In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

```
In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
array([[ 4,  7,  5,  6],
```

```
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

数组转置和轴对换

转置是重塑的一种特殊形式，它返回的是源数据的视图（不会进行任何复制操作）

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

利用np.dot计算矩阵内积

```
In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])
```

```
In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

对于高维数组，transpose需要得到一个由轴编号组成的元组才能对这些轴进行转置

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

```
In [135]: arr
Out[135]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0,  4],
        [ 1,  5],
```

```
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])

arr.swapaxes(0, 1)
```

通用函数(ufunc)：快速的元素级数组函数

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
Out[140]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

add或maximum接受2个数组（因此也叫二元（binary）ufunc），并返回一个结果数组

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,
```

```
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,
       -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,
       -0.6605])
```

返回浮点数数组的小数和整数部分

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,
5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,
0.6182,  0.45  ,  0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

利用数组进行数据处理

用数组表达式代替循环的做法，通常被称为矢量化。一般来说，矢量化数组运算要比等价的纯Python方式快上一两个数量级（甚至更多）。

假设我们想要在一组值（网格型）上计算函数sqrt(x^2+y^2)

np.meshgrid函数接受两个一维数组，并产生两个二维矩阵（对应于两个数组中所有的(x,y)

对)

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [156]: xs, ys = np.meshgrid(points, points)
In [157]: ys
Out[157]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428, 7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

matplotlib创建了这个二维数组的可视化

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[161]: <matplotlib.colorbar.Colorbar at 0x7f715e3fa630>

In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```

```
plt.show()
```

将条件逻辑表述为数组运算

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

当cond中的值为True时，选取xarr的值，否则从yarr中选取

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

假设有一个由随机数据组成的矩阵，你希望将所有正值替换为2，将所有负值替换为－2

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
```

```
       [ True, False,  True,  True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

用常数2替换arr中所有正的值

```
In [176]: np.where(arr > 0, 2, arr)
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

数学和统计方法

可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。sum、mean以及标准差std等聚合计算（aggregation，通常叫做约简（reduction））

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253
```

```
In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

arr.mean(1)是"计算行的平均值"，arr.sum(0)是"计算每列的和"

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# 所有元素的累积和
In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

```
# 所有元素的累积积
In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

用于布尔型数组的方法

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum()
Out[191]: 42
```

any用于测试数组中是否存在一个或多个True，而all则检查数组中所有值是否都是True, 这两个方法也能用于非布尔型数组，所有非0元素将会被当做True

```
In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

排序

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938,  1.24  , -0.1357,  1.43  , -0.8469])

In [197]: arr.sort()
```

```
In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24  ,  1.43  ])
```

多维数组可以在任何一个轴向上进行排序

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

顶级方法np.sort返回的是数组的已排序副本，而就地排序则会修改数组本身

唯一化以及其它的集合逻辑

找出数组中的唯一值并返回已排序的结果

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
```

```
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

函数np.in1d用于测试一个数组中的值在另一个数组中的成员资格，返回一个布尔型数组

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

表4-6：数组的集合运算

方法	说明
unique(x)	计算x中的唯一元素，并返回有序结果
intersect1d(x, y)	计算x和y中的公共元素，并返回有序结果
union1d(x, y)	计算x和y的并集，并返回有序结果
in1d(x, y)	得到一个表示“x的元素是否包含于y”的布尔型数组
setdiff1d(x, y)	集合的差，即元素在x中且不在y中
setxor1d(x, y)	集合的对称差，即存在于一个数组中但不同时存在于两个数组中的元素[译注2]

用于数组的文件输入输出

NumPy的内置二进制格式读写

np.save和np.load是读写磁盘数组数据的两个主要函数。默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为.npy的文件中的

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
```

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

通过np.savez可以将多个数组保存到一个未压缩文件中

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)

In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

将数据压缩，可以使用numpy.savez_compressed

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

线性代数

矩阵乘法的dot函数

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [226]: y
Out[226]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])
```

```
In [227]: x.dot(y)
Out[227]:
array([[  28.,   64.],
       [  67.,  181.]])
```

一个二维数组跟一个大小合适的一维数组的矩阵点积运算之后将会得到一个一维数组

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([  6.,  15.])
```

@符也可以用作中缀运算符，进行矩阵乘法

```
In [230]: x @ np.ones(3)
Out[230]: array([  6.,  15.])
```

伪随机数生成

用normal来得到一个标准正态分布的4×4样本数组

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

Python内置的random模块则只能一次生成一个样本值。从下面的测试结果中可以看出，如果需要产生大量样本值，numpy.random快了不止一个数量级

```
In [240]: from random import normalvariate
```

```
In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

numpy.random的数据生成函数使用了全局的随机种子。要避免全局状态，你可以使用numpy.random.RandomState，创建一个与其它隔离的随机数生成器

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
       -0.6365,  0.0157, -2.2427])
```

表4-8：部分numpy.random函数

函数	说明
seed	确定随机数生成器的种子
permutation	返回一个序列的随机排列或返回一个随机排列的范围
shuffle	对一个序列就地随机排列
rand	产生均匀分布的样本值
randint	从给定的上下限范围内随机选取整数
randn	产生正态分布（平均值为0，标准差为1）的样本值，类似于MATLAB接口
binomial	产生二项分布的样本值
normal	产生正态（高斯）分布的样本值
beta	产生Beta分布的样本值

表4-8：部分numpy.random函数（续）

函数	说明
chisquare	产生卡方分布的样本值
gamma	产生Gamma分布的样本值
uniform	产生在[0, 1)中均匀分布的样本值

示例：随机漫步

用np.random模块一次性随机产生1000个“掷硬币”结果（即两个数中任选一个），将其分别设置为1或－1，然后计算累计和

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps)

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()

In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

我们想要知道本次随机漫步需要多久才能距离初始0点至少10步远（任一方向均可）

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```