0 numpy概述

- NumPy是Python中科学计算的基础包。
- 它是一个Python库
- 提供多维数组对象，各种派生对象（如掩码数组和矩阵
- 提供对数据/数组各种操作

1. numpy数据类型

- numpy的数据类型和c语言中的数据类型实现并不一样
- 可以理解对基础数据类型的优化和升级
- 一切为了快速处理大型/大量数据为宗旨

1.1 numpy基础数据类型

- bool_: 布尔值，用一个字节存储
- int_： 默认整型，通常是int64/int32
- intc: 整型，通常是int32/int64
- intp: 用作索引的整型，通常是int32/int64
- int8/16/32/64: 整型
- uint8/16/32/64： 无符号整型
- float_: float64的简写
- float16: 半精度浮点型， 1bit符号， 5bits指数，10bits尾数
- float32: 单精度浮点型， 1bit符号，8bits指数，23bits尾数
- float64: 双精度浮点型， 1bit符号，11bits指数，52bits尾数
- complex_: complex128
- complex64: 复数，两个32位浮点数表示
- complex128: 复数， 由两个64位浮点数表示

1.2 numpy的数据类型

- 'b': 字节型， np.dtype('b')
- 'i': 有符号整型， np.dtype('i4')就是一个 np.int32类型
- ’u': 无符号整型， np.dtype('u8')就是一个np.uint64
- 'f': 浮点型， np.dtype('f8')
- 'c': 复数浮点型
- ’S': 'a': 字符串， np.dtype('S6')
- 'U': Unicode编码字符串， n'p.dtype('U') 就是np.str_类型
- ‘V': 原生数据， 比如空或者void， np.dtype('V')就是np.void

其中如果出现<则表示低字节序(little endian), 同理>则表示高字节序

2. numpy数组基础

- 数组操作是numpy的核心
- 数组跟list类似，但数组要求内容类型必须一致，如果不一致需要进行强制转换，可能会损失精度
- 主要包括
    - 数组的属性
    - 数组的索引
    - 数组的切分
    - 数组的编写
    - 数组的拼接和分裂

2.1 数组的属性

- numpy可以有多维数组
- 属性包括：
    - 维度(ndim)
    - 形状(shape):通常可以理解成数组各个维度的长度
    - 长度(size): 数组所存储的元素的总个数
    - 数据类型(dtype): 数组的数据类型（数组要求数据必须同一类型）
    - itemsize: 每个元素的字节长度
    - nbytes： nbytes=itemsize x size

```
import numpy as np

# 生成三个数组
a1 = np.random.randint(100, size=10) # 参数siz是生成数组的shape属性
a2 = np.random.randint(100, size=(4,6))
a3 = np.random.randint(100, size=(2,5,6))

# 打印出数组a1, a2, a3的属性
for a in [a1, a2, a3]:
    print("dtype={}, ndim={}, shape={}, size={}, itemsize={}, nbytes={}".\
          format(a.dtype, a.ndim, a.shape, a.size, a.itemsize, a.nbytes))
```

```
dtype=int32, ndim=1, shape=(10,), size=10, itemsize=4, nbytes=40
dtype=int32, ndim=2, shape=(4, 6), size=24, itemsize=4, nbytes=96
dtype=int32, ndim=3, shape=(2, 5, 6), size=60, itemsize=4, nbytes=240
```

2.2 数组的创建

索引的创建我们以后会逐步讲解，这里先介绍几个特殊数组的创建方法。

```
import numpy as np

x = np.empty(5)
print("Empty array: \n", x)

x = np.zeros(10)
print("Zeros Array: \n", x)

x = np.ones(10)
print("Ones Array: \n", x)
```

```
Empty array:
 [-4.94065646e-323  0.00000000e+000  2.12199579e-314  0.00000000e+000
  0.00000000e+000]
Zeros Array:
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Ones Array:
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

2.3 数组的索引

- 数组的索引使用方式跟list一样
- 可以是正数，负数

```
# 一位数组的索引
a1 = np.random.randint(100, size=10)
print('a1=', a1)

# 数组的索引跟list一致，分正负数，意义和用法都与list一致
print("a1[1]=", a1[1])
print("a1[-1]=", a1[-1])
```

```
a1=[79 92 40 35 85 51 11  9 69 66]
a1[1]= 92
a1[-1]= 66
```

```
# 多维数组需要使用逗号隔开，其余使用方法和list类似
a1 = np.random.randint(100, size=(3,5))
print("a1 = ", a1)

# # 数组的索引跟list一致，分正负数，意义和用法都与list一致

# 得到二维数组的某一行
```

```
print("a1[1] = ", a1[1])
# 或者
print("a1[1] = ", a1[1, :])
# 或者
print("a1[-2] = ", a1[-2])


# 得到二维数组的某一列
print("a1[:,1] = ", a1[:,1])

# 得到二维数组的某一个元素
print("a1[2,2] =" , a1[2,2])
print("a1[-1,-2] = ", a1[-1,-2])
```

```
a1 =  [[97  1 65 92 81]
 [40 75 66 53 53]
 [34 56 63 64 92]]
a1[1] =  [40 75 66 53 53]
a1[1] =  [40 75 66 53 53]
a1[-2] =  [40 75 66 53 53]
a1[:,1] =  [ 1 75 56]
a1[2,2] = 63
a1[-1,-2] =  64
```

2.4 数组的切片

- 数组的切片功能和使用同list基本一致
- 数组的切片产生的是数据的视图而不是副本，及切片后的数据和源数据是一份
- 基本用法是 x[start : stop : step] 三个参数决定切片的结果，有些可省略
    - start=0
    - stop=维度大小
    - step=1

2.4.1 一维数组的切片

- 一维数组跟list使用方法一致

```
# 一维数组的切片跟list一致，包括用法

# 定义一维数组
import numpy as np
a = np.arange(1, 11)
print("a = ", a)

# 获取中间元素
print("a[3:7] = ", a[3:7])
```

```
# 获取偶数
print("a[1::2] = ", a[1::2])

# 获取前五个元素
print("a[:5] = ", a[: 5])

# 获取后五个元素
print("a[5:] = ", a[5:])

# 后五个元素倒序
print("a[:-6:-1] = ", a[:-6:-1])
```

```
a =  [ 1  2  3  4  5  6  7  8  9 10]
a[3:7] =  [4 5 6 7]
a[1::2] =  [ 2  4  6  8 10]
a[:5] =  [1 2 3 4 5]
a[5:] =  [ 6  7  8  9 10]
a[:-6:-1] =  [10  9  8  7  6]
```

2.4.2 多维数组的切片

- 多维数组的切片跟list逻辑一致
- 使用方法上可以看作多维的list

```
# 定义多维数组
a = np.random.randint(10, size=(3,5))

#打印出a
print("a = ", "\n", a)

# 截取多维数组的一段
print("a[1:4, 1:4] = \n", a[1:4, 1:4])

# step参数也可以是负数，标识逆序
print("a[1:4:-1, 1:4:-1] = ", a[1:4:-1, 1:4:-1])
print("a[1:4:-1, 1:4:-1] = \r\n", a[::-1, ::-1])
```

```
a =
 [[1 8 5 3 3]
 [6 3 8 3 9]
 [8 9 5 2 5]]
a[1:4, 1:4] =
 [[3 8 3]
 [9 5 2]]
a[1:4:-1, 1:4:-1] =  []
a[1:4:-1, 1:4:-1] =
 [[5 2 5 9 8]
 [9 3 8 3 6]
 [3 3 5 8 1]]
```

- 获取数组的单行或者单列是一种常见操作，可以利用索引和切片技术实现这个功能。
- 处于语法简洁考虑，空的切片可以省略

```
# 利用上面
# 定义多维数组
a = np.random.randint(10, size=(3,5))

print("a = \n", a)

# 得到第二行
print("a[1, :] = ", a[1,:])

# 得到倒序的第二列
print("a[::-1, 1] = ", a[::-1, 1])

# 空的切片可以省略
# a[0, :]可以省略
print("a[0] = ", a[0])
```

```
a =
 [[1 5 1 7 0]
 [8 3 4 0 6]
 [1 2 6 9 5]]
a[1, :] =  [8 3 4 0 6]
a[::-1, 1] =  [2 3 5]
a[0] =  [1 5 1 7 0]
```

2.4.3 数组切片产生的视图和副本

- 数组切片和list切片不同之处是，数组切片的结果是源数据的视图而不是副本
- 如果想要产生源数据的副本，需要用到copy函数

```
# 证明数组切片产生的是视图而不是副本

# 1. 产生一个数组
a1 = np.zeros(shape=(3,5))
print("a1 = ",a1)

# 2. 切片产生新数组
a2 = a1[1:, 1:4]

# 3, 修改切片后数组内容
a2[1, 2] = 99

# 4. 查看结果
print("a1 = \n", a1)
print("a2 = \n", a2)

# 5. 结果证明，仅仅修改切片后数据，源数据也发生了改变
# 说明，切片仅仅是视图而不是数据的副本
```

```
a1 =  [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
a1 =
 [[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0. 99.  0.]]
a2 =
 [[ 0.  0.  0.]
 [ 0.  0. 99.]]
```

```
# 如果需要产生数据的副本，需要会用到copy功能


# 1. 产生一个数组
a1 = np.zeros(shape=(3,5))
print("a1 = ",a1)

# 2. 切片产生新数组,同时使用copy产生一个数据副本
a2 = a1[1:, 1:4].copy()

# 3, 修改切片后数组内容
a2[1, 2] = 99

# 4. 查看结果
print("a1 = \n", a1)
print("a2 = \n", a2)
```

```
a1 =  [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
a1 =
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
a2 =
 [[ 0.  0.  0.]
 [ 0.  0. 99.]]
```

2.5 数组的变形

- 变形可以通过reshape来进行操作，前提是必须前后长度一致
- 也可以通过newaxis关键字来完成
    - newaxis是一个NoneType的内容，其实就是None
    - 一般用来标识一给新的维度，比如1维的数组想变形成二维的，需要单纯的增加一个维度
    - 比如 shape=(3,) ==> shape=(3,1)

```
# shape的使用
# 需要注意数组的长度前后必须一致，此处 9=3x3
# 同时需要注意shape的参数是数组，不是几个整数参数
a = np.arange(1,10).reshape((3,3))

print("a = \n", a)
print("a.shape = ", a.shape)
```

```
a =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
a.shape =  (3, 3)
```

```
# newaxis 是单纯的增加一个维度
a = np.arange(10)
print("a = ", a)
print("a.shape = ", a.shape)

# 获得一个行向量
a1 = a[np.newaxis, :]
print("a1 = ", a1)
```

```
print("a1.shape = ", a1.shape)

# 获得一个列向量
a2 = a[:, np.newaxis]
print("a2 = ", a2)
print("a2.shape = ", a2.shape)
```

```
a =  [0 1 2 3 4 5 6 7 8 9]
a.shape =  (10,)
a1 =  [[0 1 2 3 4 5 6 7 8 9]]
a1.shape =  (1, 10)
a2 =  [[0]
 [1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
a2.shape =  (10, 1)
```

2.6 数组的拼接和分裂

本节主要研究如何把多个数组拼接成一个数组或者把一个数组拆分成多个数组的情况。

从本节开始需要注意维度的概念或者轴的概念，我们在对数组进行拼接的时候需要明确
沿着哪个轴进行拼接，以二维数组为例，这里竖轴作为第一个轴，索引值为0，横轴作为第二个轴，索引值为1，
如果默认的拼接，则默认沿着0轴。

2.6.1 数组的拼接

对数组的拼接一般默认前提是可以进行拼接，比如3x3的数组和5x5的数组拼接就可能存在问题，因为二者需要
拼接的维度是数值不等，此时强行拼接除非进行自动补齐缺失的值。

数组的拼接一般需要指明沿着哪个轴进行拼接，具体实现主要由下面几个函数完成：

- numpy.concatenate: 可以指明拼接的轴
- numpy.hstack: 沿着横轴进行拼接
- numpy.vstack: 沿着竖轴进行拼接
- numpy.dstack: 沿着第三个轴进行拼接

```
# concatenate
```

```
# 对一维数组的拼接

# 生成数组a1， a2后将两个数组拼接成a3
# 此处拼接是自己在后面自动追加
a1 = np.arange(5)
a2 = np.arange(5, 10)
a3 = np.concatenate([a1, a2])

# concatenate可以一次性链接好几个数组
a4 = np.concatenate([a1, a2, a3])

print("a1 = ", a1)
print("a2 = ", a2)
print("a3 = ", a3)
print("a4 = ", a4)
```

```
a1 =  [0 1 2 3 4]
a2 =  [5 6 7 8 9]
a3 =  [0 1 2 3 4 5 6 7 8 9]
a4 =  [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9]
```

```
# 同样，concatenate可以对多维数组进行拼接
# 拼接的时候，往往需要指出沿着哪个轴或者维度进行拼接

a1 = np.arange(12).reshape((3,4))
print("a1 = \n", a1)

a2 = np.arange(100,112).reshape((3,4))
print("a2 = \n", a2)

# 此时明显看出concatenate默认是沿着第一个轴（index=0）拼接
a3 = np.concatenate([a1, a2])
print("a3 = \n", a3)

# 如果有必要，需要指明进行拼接的轴
# 下面案例沿着第二个轴进行拼接，index=1
a4 = np.concatenate([a1, a2], axis=1)
print("a4 = \n", a4)
```

```
a1 =
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
a2 =
 [[100 101 102 103]
 [104 105 106 107]
```

```
 [108 109 110 111]]
a3 =
 [[  0   1   2   3]
 [  4   5   6   7]
 [  8   9  10  11]
 [100 101 102 103]
 [104 105 106 107]
 [108 109 110 111]]
a4 =
 [[  0   1   2   3 100 101 102 103]
 [  4   5   6   7 104 105 106 107]
 [  8   9  10  11 108 109 110 111]]
```

沿着指定的横轴或者竖轴进行拼接可以直接是红vstack或者hstack，相对简洁

```
# vstack & hstack
a1 = np.arange(4).reshape((1,4))
a2 = np.arange(100,112).reshape((3,4))
a3 = np.arange(10,13).reshape((3, 1))

# 注意vstack参数的shape
a4 = np.vstack([a1, a2])
print("a4 = \n", a4)

a5 = np.hstack([a2, a3])
print("a5 = \n", a5)
```

```
a4 =
 [[  0   1   2   3]
 [100 101 102 103]
 [104 105 106 107]
 [108 109 110 111]]
a5 =
 [[100 101 102 103  10]
 [104 105 106 107  11]
 [108 109 110 111  12]]
```

2.6.2 数组的分裂

数组的分裂研究的是如何把一个数组分割成多个数组。

主要有一下几个函数完成:

- numpy.split: 默认沿着axis=0进行分裂，可以指定轴向
- numpy.hsplit： 沿着横轴进行分裂
- numpy.vsplit： 沿着竖轴进行分裂

```
# 生成一个6x8的数组
```

```python
a1 = np.arange(48).reshape((6,8))
print("a1 = \n", a1)
print("-" * 30)

# 沿着0轴进行分裂成两部分
a2, a3 = np.split(a1, [3])
print("a2 = \n", a2)
print("-" * 30)
print("a3 = \n", a3)
print("-" * 30)

# 沿着0轴分裂成三部分
a4, a5, a6 = np.split(a1, [2,4])
print("a4 = \n", a4)
print("-" * 30)
print("a5 = \n", a5)
print("-" * 30)
print("a6 = \n", a6)
print("-" * 30)

# 沿着1轴分裂成三部分
a7, a8, a9 = np.split(a1, [2,5], axis=1)
print("a7 = \n", a7)
print("-" * 30)
print("a8 = \n", a8)
print("-" * 30)
print("a9 = \n", a9)
```

```
a1 =
 [[ 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]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]]
------------------------------
a2 =
 [[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]]
------------------------------
a3 =
 [[24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]]
------------------------------
a4 =
 [[ 0  1  2  3  4  5  6  7]
```

```
 [ 8  9 10 11 12 13 14 15]]
------------------------------
a5 =
 [[16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
------------------------------
a6 =
 [[32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]]
------------------------------
a7 =
 [[ 0  1]
 [ 8  9]
 [16 17]
 [24 25]
 [32 33]
 [40 41]]
------------------------------
a8 =
 [[ 2  3  4]
 [10 11 12]
 [18 19 20]
 [26 27 28]
 [34 35 36]
 [42 43 44]]
------------------------------
a9 =
 [[ 5  6  7]
 [13 14 15]
 [21 22 23]
 [29 30 31]
 [37 38 39]
 [45 46 47]]
```

```python
# 类似hstack之类的， vsplit，hsplit，dsplit是沿着0轴，1轴，2轴分裂的缩写

a = np.arange(20).reshape([4,5])
print("a = \n", a)

# vsplit
a1, a2 = np.vsplit(a, [2])
print("a1 = \n", a1)
print("a2 = \n", a2)

# hsplit
a3, a4, a5 = np.hsplit(a, [2,4])
print("a3 = \n", a3)
```

```
print("a4 = \n", a4)
print("a5 = \n", a5)
```

```
a =
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
a1 =
 [[0 1 2 3 4]
 [5 6 7 8 9]]
a2 =
 [[10 11 12 13 14]
 [15 16 17 18 19]]
a3 =
 [[ 0  1]
 [ 5  6]
 [10 11]
 [15 16]]
a4 =
 [[ 2  3]
 [ 7  8]
 [12 13]
 [17 18]]
a5 =
 [[ 4]
 [ 9]
 [14]
 [19]]
```

3. numpy通用函数(ufunc)

通用函数就是能同时对元素内所有元素逐个进行运算的函数。

numpy专注于大量数据运算，python本身也能够对大量数据进行计算，但是速度相对缓慢，为了解决这个问题，numpy对数据运算进行优化，使计算变得迅速简洁。

numpy进行快速数据运算的关键在于向量化。

numpy支持运算符操作，运算符看作是运算类函数的简写。

从运算符参与运算数据的角度分类，通用函数非为两类：

- 一元通用函数(unary ufunc): 对单个输入进行操作
- 二元通用函数(binary ufunc): 对两个输入进行操作

从功能上分类，通用函数分为算术计算函数，双曲三角函数，位运算类，比较运算符，弧度角度转换类等。

更加复杂的通用函数放在scipy.special模块下，如果需要，可以查阅相关文档。

3.1 通用函数-算数操作类

常见的算数运算操作如下：

- +：np.add
- -: np.subtract
- -: np.negative
- *: np.multiply
- /: np.divide
- //: np.floor_divide
- **: np.power
- %: np.mod
- absolute(abs):绝对值，比较特殊，可以处理复数，当求复数的绝对值的时候，结果是复数的幅度

```
# 运算符举例
# 需要注意的是逐个元素运算

a = np.arange(20).reshape([4,5])
print("a = \n", a)

print("a+3 = \n", a + 3)
print("a//5 = \n", a  // 3)
print("a**2 = \n", a ** 2)
print("-a = \n",  -a)
print("a % 3 = \n", a % 3)
```

```
a =
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
a+3 =
 [[ 3  4  5  6  7]
 [ 8  9 10 11 12]
 [13 14 15 16 17]
 [18 19 20 21 22]]
a//5 =
 [[0 0 0 1 1]
 [1 2 2 2 3]
 [3 3 4 4 4]
 [5 5 5 6 6]]
a**2 =
 [[  0   1   4   9  16]
 [ 25  36  49  64  81]
 [100 121 144 169 196]
 [225 256 289 324 361]]
```

```
-a =
 [[  0  -1  -2  -3  -4]
 [ -5  -6  -7  -8  -9]
 [-10 -11 -12 -13 -14]
 [-15 -16 -17 -18 -19]]
a % 3 =
 [[0 1 2 0 1]
 [2 0 1 2 0]
 [1 2 0 1 2]
 [0 1 2 0 1]]
```

absolute函数用来求绝对值，abs是缩写形式。

需要注意的是，对复数求绝对值的时候，得到的是复数的幅度值。

```
# abs举例

a = np.arange(-10,0).reshape([2,5])
print("a = \n", a)

print("abs(a) = \n", abs(a))

a = np.array([1-2j, 3-4j, 5-6j, 7-9j])
print("abs(a) = ", abs(a))
```

```
a =
 [[-10  -9  -8  -7  -6]
 [ -5  -4  -3  -2  -1]]
abs(a) =
 [[10  9  8  7  6]
 [ 5  4  3  2  1]]
abs(a) =  [ 2.23606798  5.          7.81024968 11.40175425]
```

三角函数分为正三角函数和反三角函数。

进行反三角函数的求值的时候，我这里会得到一个值错误警告，但是不会报错。

```
# 三角函数举例

theta = np.linspace(0, np.pi, 5)
print("theta = ", theta)

# 三角函数
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))

# 反三角函数
```

```
print("arcsin(theta) = ", np.arcsin(theta))
print("arccos(theta) = ", np.arccos(theta))
print("arctan(theta) = ", np.arctan(theta))
```

```
theta =  [0.         0.78539816 1.57079633 2.35619449 3.14159265]
sin(theta) =  [0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
cos(theta) =  [ 1.00000000e+00  7.07106781e-01  6.12323400e-17 -7.07106781e-01
 -1.00000000e+00]
tan(theta) =  [ 0.00000000e+00  1.00000000e+00  1.63312394e+16 -1.00000000e+00
 -1.22464680e-16]
arcsin(theta) =  [0.         0.90333911        nan        nan        nan]
arccos(theta) =  [1.57079633 0.66745722        nan        nan        nan]
arctan(theta) =  [0.         0.66577375 1.00388482 1.16942282 1.26262726]
```

```
C:\Users\augs\Anaconda3\lib\site-packages\ipykernel_launcher.py:12:
RuntimeWarning: invalid value encountered in arcsin
  if sys.path[0] == '':
C:\Users\augs\Anaconda3\lib\site-packages\ipykernel_launcher.py:13:
RuntimeWarning: invalid value encountered in arccos
  del sys.path[0]
```

指数和对数函数常用的是以e, 2, 10为底的运算，同样对于非常小的值输入，numpy也给出了精度好的运算方式。

```
a = np.array([1,2,3])
print("a = ",a)
print("e^a = ",np.exp(a))
print("2^a = ",np.exp2(a))
# 直接使用power函数进行操作
print("3^a = ",np.power(3, a))

print("ln(a) = ", np.log(a))
print("log2(a) = ", np.log2(a))
print("log10(a) = ", np.log10(a))
```

```
a =  [1 2 3]
e^a =  [ 2.71828183  7.3890561  20.08553692]
2^a =  [2. 4. 8.]
3^a =  [ 3  9 27]
ln(a) =  [0.         0.69314718 1.09861229]
log2(a) =  [0.        1.        1.5849625]
log10(a) =  [0.         0.30103    0.47712125]
```

以下特殊的版本，对非常小的输入值，能保持比较好的精度。

```
a = np.array([0, 0.001, 0.01, 0.1])
print("exp(a) - 1 = ", np.expm1(a))
print("log(1+x) = ", np.log1p(a))
```

```
exp(a) - 1 =  [0.         0.0010005  0.01005017 0.10517092]
log(1+x) =  [0.         0.0009995  0.00995033 0.09531018]
```

```
# 对exp和expm1在极小数值上的比较
print("expm1 = ", np.expm1(1e-10))
print("exp-1 = ",  np.exp(1e-10) - 1)

# log1p和log(1+x)的比较
print("log1p = ", np.log1p(1e-99))
print("log(1+x) = ", np.log(1 + 1e-99))
```

```
expm1 =  1.00000000005e-10
exp-1 =  1.000000082740371e-10
log1p =  1e-99
log(1+x) =  0.0
```

3.2 通用函数-比较类操作

此类比较操作也是逐个元素操作，最后的结果是一个包含布尔值的数组，数组的shape，size等同原数组一致。

此类操作包含：

- ==: np.equal
- !=: no.not_equal
- <: np.less
- <=: np.less_equal
- >: np.greater
- >=: np.greater_equal

```
# 比较类操作案例
a = np.random.randint(100, size=(2,5))
print("a = \n", a)

print("a < 50 : \n", a < 50)
print("a == 50 : \n", a == 50)
```

```
a =
 [[67 14 95 28 51]
 [80  5 65 41  0]]
a < 50 :
 [[False  True False  True False]
 [False  True False  True  True]]
a == 50 :
 [[False False False False False]
 [False False False False False]]
```

```
# 常用的np关于比较运算的操作
# count_nonzero用来统计非零的值个数
a = np.random.randint(100, size=(2,5))
print("a = \n", a)

print()

# count_nonzero用来统计非零的值个数
# 统计小于50的数字的个数
print("小于50的格式总共 {}个".format(np.count_nonzero(a < 50)))

#或者
print()

# 布尔值也可以作为数字运算，所以可以直接求和
# 统计大于50的数字的个数
print("大于50的格式总共 {}个".format(np.sum(a > 50)))
```

```
a =
 [[52  3 57  5 50]
 [47 31 46 77 69]]

小于50的格式总共 5个

大于50的格式总共 4个
```

```
# 如果检测结果是否包含真值或者全部是否都是某个值
# 可以用any或者all
a = np.array([1,2,3,4,5,6])
print("a = ", a)

print("a中包含大于10的数字吗：", np.any(a > 10))
print("a中包含大于5的数字吗：", np.any(a > 5))
print("a中的数组都大于0吗：", np.all(a > 0))
```

```
a =  [1 2 3 4 5 6]
a中包含大于10的数字吗： False
a中包含大于5的数字吗： True
a中的数组都大于0吗： True
```

3.3 通用函数-按位运算

numpy提供了可以对数组进行布尔运算的操作夫，此类操作符称为逐位运算符(bitwise logic operator)。

逐个运算符包括以下几个：

- &： np.bitwise_and
- |: np.bitwise_or
- ^: np.bitwise_xor
- ~: np.bitwise_not

```
a = np.arange(20).reshape([4,5])
print("a = \n", a)
print("a中能被3整除或者7整除的数字保留：")
print((a % 3 == 0) | (a % 7 == 0))
```

```
a =
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
a中能被3整除或者7整除的数字保留：
[[ True False False  True False]
 [False  True  True False  True]
 [False False  True False  True]
 [ True False False  True False]]
```

3.4 将布尔值作为掩码操作

通过组合使用比较类运算，可以快速提取出符合条件的数值，此类操作称为掩码操作。

```
a = np.random.randint(100, size=(3,5))
print("a = \n", a)
print()

# 掩码可以快速提取数据，比如，提取出小于50的数据
a1 = a[a<50]
print("a1 = ", a1)

print("a1.shape = ", a1.shape)
```

```
a =
 [[45 22 92  1 14]
 [17 90 96  8 61]
 [21 87 69 44 47]]

a1 =  [45 22  1 14 17  8 21 44 47]
a1.shape =  (9,)
```

3.5 通用函数-特性

numpy是为了大量数据运算而生，所以根据大量数据计算的特殊情况，有一些特殊的属性，本节我们做一个简单介绍。

3.5.1 指定输出

大量数据运算的临时结果一般放在内存变量中，但有时候可能需要保存中间结果，即把中间结果写入指定位置，此时就需要用到指定输出功能。

所有通用函数都可以带参数out，out即是需要将结果写入的位置。

```
a = np.arange(10)
b = np.empty(10)

# 此时把中间结果存入b，最终结果存入c
# 此处中间结果和最终结果一致
c = np.multiply(a, 2, out=b)
print("a = ", a)
print("b = ", b)
print("c = ", c)
```

```
a =  [0 1 2 3 4 5 6 7 8 9]
b =  [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18.]
c =  [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18.]
```

out也可以直接作用于数组的视图，可以直接更改数组内容。

下面的结果，跟直接对b赋值是由区别的：

```
>>> b[::2] = 2 ** a
```

上面代码会计算结果，将结果放入临时数组作为中间变量保存，最后作为结果复制给b[::2]，但是如果使用out
则把结果直接写入b中，减少资源使用。

虽然就下面案例来讲，并没有节省多少资源，但如果数据量特别大的时候，效果非常明显，有时甚至是必要手段。

```
# out也可以是一个数组的视图，这样可以直接更改数组内容
a = np.arange(5)
b = np.zeros(10)

np.power(2, a, out=b[::2])

print("a = ", a)
print("b = ", b)
```

```
a =  [0 1 2 3 4]
b =  [ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

3.5.2 外积

数组的外积，就是数组对应逐个元素相乘，即获得两个数组所有元素对的乘积，假定数组a1， a2，每个数组10个元素，总共获得的外积应该有 10 x 10 = 100 个元素。

```
# 外积计算
a = np.arange(1,6)
b = np.arange(10, 15)
print("a = \n", a)

print()
print("b = \n", b)
print()
c = np.multiply.outer(a, b)
print("c = \n", c)
```

```
a =
 [1 2 3 4 5]

b =
 [10 11 12 13 14]

c =
 [[10 11 12 13 14]
 [20 22 24 26 28]
 [30 33 36 39 42]
 [40 44 48 52 56]
 [50 55 60 65 70]]
```

4. 聚合

当面对大量数据的时候，我们经常需要可计算数据的一些统计方面的信息，所幸numpy给我们提供了很多好用的聚合类功能，可以让我们方便的进行一些操作。

numpy的聚合可以直接作用在数组上面，不需要每次都numpy.xxx调用。
同时一般如果名称相同功能也相同，相对速度比python同名功能要快。

numpy中可用的聚合函数如下，需要注意的是，除any和all之外，每个函数都存在一个NaN安全的版本，形式是在函数名称前加nan,例如np.nansum就是sum的NaN安全版本：

- np.sum: 和
- np.prod: 积
- np.mean: 平均数
- np.std: 标准差
- np.var: 方差
- np.min: 最小值
- np.max: 最大值
- np.argin: 最小值的索引
- np.argmax: 最大值的索引
- np.median: 中位数
- mp.oercentile: 基于元素排序的统计值
- np.any: 是否至少存在一个为真的元素
- np.all: 所有元素是否为真

下面我们给大家举例介绍：

4.1 适用通用函数的聚合函数

此类函数主要指的是reduce和accumulate。

- reduce: 功能同python的标准reduce一致，即重复执行某一个操作知道最后一个结果。
- accumulate: 计算的中间结果会被存储

```
# reduce
a = np.arange(100)

# 把a内所有值进行相加
b = np.add.reduce(a)
print("b = ", b)

# 把a内所有制相加，但相加的中间结果需要保存下来
c = np.add.accumulate(a)
print("c = \n", c)
```

```
b =  4950
c =
 [   0    1    3    6   10   15   21   28   36   45   55   66   78   91
  105  120  136  153  171  190  210  231  253  276  300  325  351  378
  406  435  465  496  528  561  595  630  666  703  741  780  820  861
  903  946  990 1035 1081 1128 1176 1225 1275 1326 1378 1431 1485 1540
 1596 1653 1711 1770 1830 1891 1953 2016 2080 2145 2211 2278 2346 2415
 2485 2556 2628 2701 2775 2850 2926 3003 3081 3160 3240 3321 3403 3486
 3570 3655 3741 3828 3916 4005 4095 4186 4278 4371 4465 4560 4656 4753
 4851 4950]
```

4.2 数组值求和

numpy.sum函数和python的求和函数功能基本一致,但是还是有一些小区别:

- numpy的求和函数具有维度的概念，求和多项可以是数组
- 同时参数含义跟python的sum并不一致
- numpy.sum速度快一些

```
#
a = np.arange(100).reshape((10,10))

b = np.sum(a)
print(b)
```

```
4950
```

4.3 最大值和最小值

求最大值最小值:

- min：最小值
- max：最大值

```
# 请注意一下三个求最小值的区别

a = np.random.rand(20)
print("a = \n", a)

# 调用pyhton的标准min
m1 = min(a)
print("最小值:  ", m1)

# 调用numpy的min
m2 = np.min(a)
print("最小值:  ", m2)

# numpy.min的简写形式，跟进a的类型会自动调用numpy.min
m3 = a.min()
print("最小值:  ", m3)
```

```
a =
 [0.23934032 0.15596611 0.85993783 0.13911453 0.03351837 0.73838562
 0.21018289 0.72901198 0.94990497 0.980913   0.40451059 0.22112041
 0.29087158 0.86319102 0.30870189 0.53390399 0.74920732 0.00546603
 0.60973917 0.15531549]
最小值:  0.005466033221787847
最小值:  0.005466033221787847
最小值:  0.005466033221787847
```

4.4 多维度聚合

numpy研究的数据经常是多个维度的，这时候经常需要的操作是沿着某一轴进行操作，此时，聚合函数都会有一个参数axis，用来表示需要沿着哪个轴进行聚合。

```
a = np.random.randint(100, size=(4,5))
print("a = \n", a)

# 求最大值，每一行
b = a.max(axis=1)
print("a每一行的最大值是:  ", b)

# 求每一列的和
c = a.sum(axis=0)
print("a每一列的和是: ", c)
```

```
a =
 [[48 17  0 54 12]
 [76  1 69 14 36]
 [77 44 90 32 50]
 [44 45  8  9 68]]
a每一行的最大值是：  [54 76 90 68]
a每一列的和是：  [245 107 167 109 166]
```

5. 广播

对不同大小的数组进行计算的时候，需要想法对齐数组的长度，广播就是自动对齐数组长度的一种规则。

5.1 广播的介绍

广播允许对不同大小的数组进行操作，例如前面介绍的一个标量和一个数组相加，这种自动把自己编程和对方形状一样然后进行
操作的能力，叫广播。

```
a = np.zeros(5)
print("a = ", a)

# 我们可以认为是 a + 4 = array(0,0,0,0,0) + array(4,4,4,4,4)
# 此时标量4自动扩展成了 array(4,4,4,4,4)
b = a + 4
print("b = ", b)
```

```
a =  [0. 0. 0. 0. 0.]
b =  [4. 4. 4. 4. 4.]
```

```
a = np.arange(5)
print("a = ", a)
print()

b = np.arange(15).reshape((3,5))
print("b = \n", b)
print()

# 此处相当于把b按行扩展成了一个 3x5的数组，然后和a进行bitwise的相加
c = a + b
print("c = \n", c)
```

```
a =  [0 1 2 3 4]

b =
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

c =
 [[ 0  2  4  6  8]
 [ 5  7  9 11 13]
 [10 12 14 16 18]]
```

```
# 负责例子，需要两边广播

a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

c = a + b
print("a = \n", a)
print()
print("b = \n", b)
print()
print("c = \n", a + b)
```

```
a =
 [0 1 2]

b =
 [[0]
 [1]
 [2]]

c =
 [[0 1 2]
 [1 2 3]
 [2 3 4]]
```

5.2 广播的规则

广播必须按照一定规则进行，不能瞎jb播，即便是按照规则，也不是一定能进行广播。

广播规则如下：

- 规则1： 如果两个数组维度不相同，则小维度数组形状在最左边补上1
- 规则2： 如果两数组的形状在任何一个维度上都不匹配，则数组的形状会沿着维度为1的维度扩展以匹配零位一个数组

- 规则3： 如果两两个数组的形状在任何一个维度上都不匹配并且没有任何一个维度等于1， 则异常。

```
# 规则1 案例

a = np.arange(15).reshape((3,5))
b = np.arange(5)

c = a + b

print("a.shape = ", a.shape)
print("a = \n", a)
print()

print("b.shape = ", b.shape)
print("b = \n", b)
print()

print("c.shape = ", c.shape)
print("c = \n", c)
```

```
a.shape =  (3, 5)
a =
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

b.shape =  (5,)
b =
 [0 1 2 3 4]

c.shape =  (3, 5)
c =
 [[ 0  2  4  6  8]
 [ 5  7  9 11 13]
 [10 12 14 16 18]]
```

上面例子是规则1的运用案例。

```
a.shape = (3,5)
b.shape = (5,)
```

此时如果a, b相加，则按照广播规则1， 需要在b.shape的最左边补上1， 即：

```
b.shape=(5,) ==> b.shape=(1,5)
```

然后，可以看作再把b.shape进行第二次扩展：

```
b.shape(1,5) ==> b.shape(3,5)
```

```
# 广播规则2
a = np.arange(5).reshape((5,1))
b = np.arange(5)

c = a + b

print("a.shape = ", a.shape)
print("a = \n", a)
print()

print("b.shape = ", b.shape)
print("b = \n", b)
print()

print("c.shape = ", c.shape)
print("c = \n", c)
```

```
a.shape =  (5, 1)
a =
 [[0]
 [1]
 [2]
 [3]
 [4]]

b.shape =  (5,)
b =
 [0 1 2 3 4]

c.shape =  (5, 5)
c =
 [[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]]
```

上面例子按照规则2进行挂广播。
先考察a, b的shape

```
a.shape = (5, 1)
b.shape = (5, )
```

此时广播需要先对b进行扩展：

```
b.shape=(5, ) ==> (1,5)
```

然后跟进a和b的shape分别进行扩展：

```
a.shape=(5,1) ==> (5,5)
b.shape=(1,5) ==> (5,5)
```

```
# 不能广播的案例

a = np.arange(15).reshape((3,5))
b = np.arange(3)

print("a.shape = ", a.shape)
print("b.shape = ", b.shape)

c = a + b
```

```
a.shape =  (3, 5)
b.shape =  (3,)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-95-081747746e13> in <module>()
      7 print("b.shape = ", b.shape)
      8
----> 9 c = a + b
```

```
ValueError: operands could not be broadcast together with shapes (3,5) (3,)
```

以上案例不能进行广播。
我们考察a和b的shaple：

```
a.shape = (3,5)
b.shape = (3, )
```

我们如果需要广播，则按照广播的规则，需要先把b的shape进行补足,则按照规则1：

```
b.shape = (3,) ==> (1,3)
```

经过规则1的补齐后，a，b的shap而变成了如下：

```
a.shape = (3,5)
b.shape = (1,3)
```

然后按照规则2再次进行匹配后的结果是：

```
a.shape = (3,5)
b.shape = (3,3)
```

发现经过匹配后，还是不一致，匹配失败！！！

6. 花哨的索引

- 使用数组作为索引叫花哨的索引(Fnacy Indexing)
- 花哨的索引让我们快速访问复杂数组的子数据集
- 使用花哨的索引同样可以对子数据集进行写操作
- 利用花哨的索引获得的结果与索引的形状(Shape)一致，跟被索引的数组的形状无关。

```
import numpy as np

# 产生10个随机数字
x = np.random.randint(100, size=10)
print("x = ", x)

indx = [2,3,7]

#用数组作为索引就是花哨的索引
a = x[indx]
print("a = ",a)

# 结果的shape跟索引的shape一致
print("a.shape = ", a.shape)
```

```
x =  [23 42 29 74 57 58 53 44 11  6]
a =  [29 74 44]
a.shape =  (3,)
```

```
indx = np.array([2,3,5,7]).reshape((2,2))
print("indx.shape = ", indx.shape)

print()
# 使用一个2x2的数组作为索引
b = x[indx]
print("b = ", b)

print()
# 结果的shape跟索引的shape一致
print("b.shape = ", b.shape)
```

```
indx.shape =  (2, 2)

b =  [[29 74]
 [58 44]]

b.shape =  (2, 2)
```

对于花哨的索引，可以使用两个数组分别表示，但是在索引的配对的时候，需要遵守广播规则才能一对一配对，例如下面例子：

```
# 花哨的索引还可以更花哨

x = np.arange(16).reshape((4,4))
print("x = ", x)
print()

print("x.shape = ", x.shape)

print()

# 我们会获得的结果是三个数字组成的数组
# 三个数字分别是(0,3),  (2,1), (3,1)
r = np.array([0,2,3])
c = np.array([3,1,1])

a = x[r, c]
print("a = ", a)
print("a.shape = ",a.shape)
```

```
x =  [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

x.shape =  (4, 4)

a =  [ 3  9 13]
a.shape =  (3,)
```

花哨的索引还可以有更花哨的用法，比如:

```
import numpy as np
x = np.arange(20).reshape((4,5))
print("x = ", x)
print("x.shape = ", x.shape)

print()
```

```
x =  [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
x.shape =  (4, 5)
```

```
# 1.  简单的索引和花哨的索引组合使用你

a = x[2, [3,2,1]]
print("x[2, [3,2,1]] = ", a)
print("a.shape = ", a.shape)
```

```
x[2, [3,2,1]] =  [13 12 11]
a.shape =  (3,)
```

```
# 2, 花哨的索引+切片配合服用
print()
b = x[2:, [3,2,1]]
print("x[2:, [3,2,1]] = ", b)
print("b.shape = ", b.shape)
```

```
x[2:, [3,2,1]] =  [[13 12 11]
 [18 17 16]]
b.shape =  (2, 3)
```

```
# 3. 花哨的索引+掩码

print(x)
mask = np.array([1,0,1,1,0], dtype=bool)


print(c)
c = x[[0,2,3], mask]

print("x[[0,2,3], mask] = \n", c)
print("c.shape = ", c.shape)
```

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

[ 0 12 18]
x[[0,2,3], mask] =
 [ 0 12 18]
c.shape =  (3,)
```

```
# 利用花哨的索引批量修改数据

x = np.arange(10)
x[[2,4,6,8]] = 999

print(x)
```

```
[  0   1 999   3 999   5 999   7 999   9]
```

7. 数组的排序

- 数组排序默认使用快排
- 根据选择，可以选用其他方式排序
- 可选参数还有axis，可以选择沿着哪个方向排序，此时把这个方向的数据当初独立数组对待

7.1 sort

- 默认快排, 其他归并排序，堆排序， 稳定排序
- 不修改源数据，返回排好的数据下使用np.sort(data)
- 修改数据源，使用 data.sort()

```
x = np.random.randint(20, size =10)
print("排序前 x = ", x)



a = np.sort(x)
print("排序后 x = ", x)
print("排序后 a = ", a)

x.sort()
print("修改数据源排序后 x = ", x)
```

```
排序前 x =  [ 3  8  5  7 17  0 16 16  1  8]
排序后 x =  [ 3  8  5  7 17  0 16 16  1  8]
排序后 a =  [ 0  1  3  5  7  8  8 16 16 17]
修改数据源排序后 x =  [ 0  1  3  5  7  8  8 16 16 17]
```

7.2 argsort

- 返回排序后数据的索引

```
x = np.random.randint(20, size =10)
print("排序前 x = ", x)

a = np.argsort(x)
print("排序后的索引：  ", a)
```

```
排序前 x =  [10  2  0  6 10  3  9  2 18  8]
排序后的索引：   [2 1 7 5 3 9 6 0 4 8]
```

7.3 分隔

- 选择出前k个最小的值,以排序第k个值为界限，小于它的在前面，等于大于它的在后面
- 选择结果在数组左侧，其余的在数组右侧
- 两侧排序不规则
- 使用方法是 np.partition(x, k)

```
x = np.random.randint(100, size=10)
print("排序前 x= ", x)

a = np.partition(x, 4)
print("分隔后的值 x= ", a)
```

```
排序前 x=  [99 71 55 60 86 16 18 51 53 37]
分隔后的值 x=  [16 51 18 37 53 71 55 60 86 99]
```

```
x = np.random.randint(100, size=(4,5))
print("排序前 x = \n", x)

a = np.partition(x, 3, axis=1)
print("沿着axis=1分隔后的数据是 \n", a)
```

```
排序前 x =
 [[33 16  2 12 51]
 [28 61 55 95 18]
 [30 23 72 57 89]
 [20 49 23 26  0]]
沿着axis=1分隔后的数据是
 [[12  2 16 33 51]
 [18 28 55 61 95]
 [57 30 23 72 89]
 [ 0 20 23 26 49]]
```

8. 结构化数据

- 用于存储异构数值，类似c语言的结构
- 复杂结构建议使用Pandas的DataFrame

8.1 结构化数组初体验

```
# 定义一个学生信息数组，包括姓名，成绩，身高
x = np.zeros(4, dtype={"names":("name", "score", "heigh"), \
                       "formats":("U10", "i4", "f8")})

print("x = ", x)
print("x.dtype = ", x.dtype)
```

```
x =  [('', 0, 0.) ('', 0, 0.) ('', 0, 0.) ('', 0, 0.)]
x.dtype =  [('name', '<U10'), ('score', '<i4'), ('heigh', '<f8')]
```

```
# 可以对某一项进行统一赋值
x['name'] = "LIU Ying"
x['score'] = 98
x['heigh'] = 185

print("统一赋值后的x = \n", x )
```

```
统一赋值后的x =
 [('LIU Ying', 98, 185.) ('LIU Ying', 98, 185.) ('LIU Ying', 98, 185.)
 ('LIU Ying', 98, 185.)]
```

```
# 可以采用key的形式进行访问列
print("学生的姓名是:  ", x['name'])

# 也可以采用索引的形式进行访问行
print("第一行的同学是: ", x[0])
```

```
学生的姓名是:  ['LIU Ying' 'LIU Ying' 'LIU Ying' 'LIU Ying']
第一行的同学是:  ('LIU Ying', 98, 185.)
```

```
# 可以对结构化数据进行批量赋值

names = ["Alice", "Bob", "Cindy", "Day"]
score = [86, 45,68,98]
heigh = [154, 184, 198,178]

x["name"] = names
x["score"] = score
x["heigh"] = heigh

print("统一赋值后的 x = \n", x)
```

```
统一赋值后的 x =
 [('Alice', 86, 154.) ('Bob', 45, 184.) ('Cindy', 68, 198.)
 ('Day', 98, 178.)]
```

```
# 结构化数据的掩码操作
# 选择成绩小于90的同学的姓名
print(x[x['score'] < 90]['name'])
```

```
['Alice' 'Bob' 'Cindy']
```

8.2 结构化数据的常用类型定义方式

结构化数组的类型的定义方式比较灵活，常用的有以下几种方式：

```
# 1. 使用字典作为机构化数组的类型

a = np.dtype({"names":("name", "score", "heigh"), \
                      "formats":("U10", "i4", "f8")})
print(a)
```

```
[('name', '<U10'), ('score', '<i4'), ('heigh', '<f8')]
```

```
# 2. 使用python数据类型或者NumPy的dtype类型

b = np.dtype({"names":("name", "score", "heigh"), \
                      "formats":((np.str_, 10), int, float)})

print(b)
```

```
[('name', '<U10'), ('score', '<i4'), ('heigh', '<f8')]
```

```
# 3. 如果类型的名称不重要， 可以省略
c = np.dtype("S10, i4, f8")
print(c)
```

```
[('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')]
```

```
# 4. 复合类型
```

```python
# 此类型类似C语言中的结构
import numpy as np

# 定义一个结构
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (4,4))])

# 利用定义的结构tp创建一份内容
x = np.zeros(2, dtype=tp)

#打印整个内容
print(x)

# 只打印出数据内容
print(x['mat'][0])
```

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