



大数据，成就未来



# NumPy 数值计算基础

# 目录

---



## 写在前面

- 假设有想计算两个维数较大的向量的点积，即  $a \cdot b$ ,
- $$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \\ a_n \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

```
1 import numpy as np
2 import time
3 a = np.random.rand(1000000)
4 b = np.random.rand(1000000)
5 c = 0
6 tic = time.time()
7 for i in range(1000000):
8     c += a[i]*b[i]
9 toc = time.time()
10 print("矩阵内积为: ", c)
11
12 print("使用循环的运行时间: %s ms" % str(1000*(toc-tic)))
```

矩阵内积为: 249859.32652219647

使用循环的运行时间: 575.9997367858887 ms

## 写在前面

- 假设有两个维数较大的向量的点积，即  $a \cdot b$ ,
- $$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \\ a_n \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

```
1 tic = time.time()
2 c = np.dot(a, b)
3 toc = time.time()
4 print("矩阵内积为: ", c)
5
6 print("使用向量点积的运行时间: %s ms" % str(1000*(toc-tic)))
```

矩阵内积为: 249859.3265222009

使用向量点积的运行时间: 2.000093460083008 ms

## 写在前面

- 假设已知一个维数较大的向量 $v$ ，想计算 $u$

```
1 import numpy as np
2 import math
3 v = np.random.rand(1000000)
4 tic = time.time()
5 u = np.zeros((1000000, 1))
6 for i in range(1000000):
7     u[i] = math.exp(v[i])
8 toc = time.time()
9 print("使用循环运行时间: ", 1000*(toc-tic))
10 tic = time.time()
11 u = np.exp(v)
12 toc = time.time()
13 print("使用向量化运行时间: ", 1000*(toc-tic))
```

使用循环运行时间: 648.9996910095215

使用向量化运行时间: 7.999897003173828

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_n \end{bmatrix} \Rightarrow \mathbf{u} = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ e^{v_3} \\ \dots \\ e^{v_n} \end{bmatrix}$$

# 创建数组对象

---

## 理解Python中的数据类型

- Python整数不仅仅是一个整数
- 标准的Python实现是用C编写的。
- 这意味着每个Python对象都只是一个巧妙伪装的C结构，它不仅包含它的值，还包含其他信息。
- 例如，当我们在Python中定义一个整数时，比如 `x = 10000`，`x`不只是一个“原始”整数。
- 它实际上是一个指向复合C结构的指针，它包含几个值。

# 创建数组对象

## 理解Python中的数据类型

➤ 查看Python 3的源代码，我们发现整型(long)类型定义实际上是这样的(在扩展C宏之后):

- ```
struct _longobject {  
    long ob_refcnt;           #一个帮助Python静默处理内存分配和释放的引用计数  
    PyTypeObject *ob_type;    #变量的类型  
    size_t ob_size;           #指定以下数据成员的大小  
    long ob_digit[1];         #包含我们期望Python变量表示的实际整数值  
};
```

# 创建数组对象

---

## 理解Python中的数据类型

- 注意这里的区别:一个C整数本质上是一个在内存中位置的标签, 它是字节编码一个整数值。
- Python整数是指向包含所有Python对象信息的内存位置的指针, 包括包含整数值的字节。
- Python整数结构中的这些额外信息使得Python能够如此自由和动态地编码。
- 然而, 在Python类型中, 所有这些附加信息都是以成本为代价的, 在组合了许多这些对象的结构中, 这一点变得尤为明显。



# 创建数组对象

## Python列表不仅仅是一个列表

- 让我们考虑一下，当我们使用一个包含许多Python对象的Python数据结构时，会发生什么情况
- Python中的标准可变多元素容器是列表。
- 我们可以创建一个整数列表如下：

```
In [1]: L = list(range(10))  
L
```

```
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [2]: type(L[0])
```

```
Out[2]: int
```

或者类似的字符串列表：

```
In [3]: L2 = [str(c) for c in L]  
L2
```

```
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
In [4]: type(L2[0])
```

```
Out[4]: str
```

# 创建数组对象

---

## Python列表不仅仅是一个列表

- 由于Python的动态类型，我们甚至可以创建异构列表:

```
In [5]: L3 = [True, "2", 3.0, 4]
         [type(item) for item in L3]
```

```
Out[5]: [bool, str, float, int]
```

# 创建数组对象

---

## Python列表不仅仅是一个列表

- 如果在所有变量都是相同类型的特殊情况下，大部分**信息是冗余的**
- 所以在**固定类型**的数组中存储数据会**更有效**。
- 在实现级别，数组本质上包含一个指向连续数据块的指针。
- 另一方面，Python列表包含指向一个指针块的指针，每个指针都指向一个完整的Python对象，就像我们前面看到的Python整数。
- **固定类型的numpy样式的数组缺少这种灵活性，但是对于存储和操作数据来说更有效。**

# 创建数组对象

## 在Python中固定数组

- Python提供了几种不同的选项，用于在高效的固定类型数据缓冲区中存储数据。
- 内置的 “array”模块(从Python 3.3开始可用)可用于创建统一类型的密集数组:

```
In [9]: import array  
L = list(range(10))  
A = array.array('i', L)  
A
```

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

这里'i' 表示内容为整数的类型代码

- 但是array模块不支持多维，也没有各种运算函数。

# 创建数组对象

## Python列表不仅仅是一个列表

- 更有用的是NumPy包的**ndarray对象**
- 虽然Python的 array 对象提供了有效存储基于数组的数据，但是NumPy为该数据添加了更高效的操作，例如数组的运算、二维数组的转置、计算向量内积、对数组进行函数运算（余弦等）。
- Numpy还是很多更高级扩展库的依赖库，如Scipy, Matplotlib, Pandas等
- 另外，Numpy内置函数处理速度是C语言级别的，所以要尽量使用内置函数
- 这里，将演示创建NumPy数组的几种方法。从标准的NumPy导入开始：

```
import numpy as np
```

<https://docs.scipy.org/doc/numpy/genindex.html>

# 创建数组对象

## 1. 数组属性：ndarray（数组N-dimensional array）是存储单一数据类型多维数组。

➤ 为了更好的理解和使用数组，要先了解数组的基本属性。

| 属性       | 说明                                      |
|----------|-----------------------------------------|
| ndim     | 返回 int。表示数组的维数（秩）                       |
| shape    | 返回 tuple。表示数组的尺寸，对于 n 行 m 列的矩阵，形状为(n,m) |
| size     | 返回 int。表示数组的元素总数，等于数组形状的乘积              |
| dtype    | 返回 data-type。描述数组中元素的类型                 |
| itemsize | 返回 int。表示数组的每个元素的大小（以字节为单位）。            |
| flags    | 返回 ndarray 对象的内存信息                      |
| real     | 返回 ndarray元素的实部                         |
| imag     | 返回 ndarray 元素的虚部                        |

# 创建数组对象

## 2. 数组创建

- Numpy提供的array函数可以创建一维或多维数组，基本语法就是：

*`numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)`*

| 参数名称                | 说明                                                    |
|---------------------|-------------------------------------------------------|
| <code>object</code> | 接收array。表示想要创建的数组。无默认。                                |
| <code>dtype</code>  | 接收data-type。表示数组所需的数据类型。如果未给定，则选择保存对象所需的最小类型。默认为None。 |
| <code>ndmin</code>  | 接收int。指定生成数组应该具有的最小维数。默认为None。                        |

# 创建数组对象

## ➤ 创建数组并查看数组属性

```
In[1]:      import numpy as np          #导入NumPy库
           arr1 = np.array([1, 2, 3, 4])  #创建一维数组
           print('创建的数组为: ',arr1)
```

```
Out[1]:      创建的数组为:  [1 2 3 4]
```

```
In[2]:      arr2 = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]]) #创建二维数组
           print('创建的数组为: \n',arr2)
```

```
Out[2]:      创建的数组为:
           [[ 1  2  3  4]
            [ 4  5  6  7]
            [ 7  8  9 10]]
```



# 创建数组对象

## ➤ 创建数组并查看数组属性

|         |                                                               |
|---------|---------------------------------------------------------------|
| In[3]:  | <pre>#查看数组类型<br/>print('数组类型为: ',arr2.dtype)</pre>            |
| Out[3]: | 数组类型为: int32                                                  |
| In[4]:  | <pre>#查看数组元素个数<br/>print('数组元素个数为: ',arr2.size)</pre>         |
| Out[4]: | 数组元素个数为: 12                                                   |
| In[5]:  | <pre>#查看数组每个元素大小<br/>print('数组每个元素大小为: ',arr2.itemsize)</pre> |
| Out[5]: | 数组每个元素大小为: 4                                                  |

创建的arr2数组为:

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

# 创建数组对象

## ➤ 查看数组结构，重新设置数组的 shape 属性

```
In[6]: #查看数组结构  
print('数组维度为: ',arr2.shape)
```

```
Out[6]: 数组维度为: (3, 4)
```

创建的arr2数组为:

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

```
In[7]: #重新设置 shape  
arr2.shape = 4,3  
print('重新设置 shape 后的 arr2 为: \n',arr2)
```

```
Out[7]: 重新设置shape维度后的arr2为:  
[[ 1  2  3]  
 [ 4  4  5]  
 [ 6  7  7]  
 [ 8  9 10]]
```

# 创建数组对象

## ➤ 创建数组并查看数组属性

```
In[1]:      import numpy as np                #导入NumPy库
           arr1 = np.array([1, 2, 3, 4])      #创建一维数组
           print('创建的数组为: ',arr1)
```

```
Out[1]:      创建的数组为:  [1 2 3 4]
```

先创建一个Python序列，然后通过array的数将其转换为数组

```
In[2]:      arr2 = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]]) #创建二维数组
           print('创建的数组为: \n',arr2)
```

```
Out[2]:      创建的数组为:
           [[ 1  2  3  4]
            [ 4  5  6  7]
            [ 7  8  9 10]]
```

效率不高!  
NumPy提供了很多专门用来创建数组的函数

# 创建数组对象

- 使用 `arange` 函数创建等差数列数组
- *`arange([start,] stop[, step,], dtype=None)`*
- `arange`有四个参数，分别是起始点、终值、步长、和返回类型。

```
In[8]:      print('使用 arange 函数创建的数组为: \n', np.arange(0, 1, 0.1))
```

```
Out[8]:      使用arange函数创建的数组为: [ 0.  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
```

# 创建数组对象

- 使用 `linspace` 函数创建等差数列数组
- *`linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)`*
- 函数通过指定开始值、终值和元素个数来创建一维数组，默认设置**包括终值**，这点要与`arange`函数区分开

```
In[9]:      print('使用 linspace 函数创建的数组为: \n', np.linspace(0, 1, 12))
```

```
Out[9]:      使用linspace函数创建的数组为:  
[0.      0.09090909 0.18181818 0.27272727 0.36363636 0.45454545  
 0.54545455 0.63636364 0.72727273 0.81818182 0.90909091 1.      ]
```

# 创建数组对象

- 使用 `logspace` 函数创建等比数列
- *`logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None, axis=0)`*
- 与 `linspace` 函数相似，它创建的是等比数列
- 起始位和终止位代表的是10的幂（默认基数为10），0代表10的0次方，9代表10的9次方

```
In[10]: print('使用logspace函数创建的数组为：', np.logspace(0, 2, 20))
```

```
Out[10]: 使用logspace函数创建的数组为： [ 1.          1.27427499  
1.62377674 ..., 61.58482111 78.47599704 100.         ]
```

# 创建数组对象

## ➤ 使用zeros函数创建数组

## ➤ *zeros(shape, dtype=float, order='C')*

```
In[11]: print('使用zeros函数创建的数组为: ', np.zeros((2,3)))
```

```
Out[11]: 使用zeros函数创建的数组为:  
[[ 0.  0.  0.]  
 [ 0.  0.  0.]
```

# 创建数组对象

## ➤ 使用eye函数创建数组

➤ *`eye(N, M=None, k=0, dtype=<class 'float'>, order='C')`*

➤ eye函数用来创建主对角线上的元素都为1，其他的元素都为0的数组，类似单位矩阵。

```
In[12]: print('使用eye函数创建的数组为: ', np.eye(3))
```

```
Out[12]: 使用eye函数创建的数组为:  
[[ 1.  0.  0.]  
 [ 0.  1.  0.]  
 [ 0.  0.  1.]]
```



# 创建数组对象

## ➤ 使用diag函数创建数组

### ➤ *diag(v, k=0)*

- diag函数创建类似对角的数组，即除了对角线以外的其他元素都是0，对角线上的元素可以是0或其他值。

```
In[13]: print('使用diag函数创建的数组为: ', np.diag([1,2,3,4]))
```

```
Out[13]: 使用diag函数创建的数组为:  
[[1 0 0 0]  
 [0 2 0 0]  
 [0 0 3 0]  
 [0 0 0 4]]
```

# 创建数组对象

- 使用ones函数创建数组
- *ones(shape, dtype=None, order='C')*
- ones函数用来创建元素全部为1的数组

```
In[14]: print('使用ones函数创建的数组为：', np.ones((5,3)) )
```

```
Out[14]: 使用ones函数创建的数组为：  
[[ 1.  1.  1.]  
 [ 1.  1.  1.]  
 [ 1.  1.  1.]  
 [ 1.  1.  1.]  
 [ 1.  1.  1.]
```

# 创建数组对象

## 3. 数组数据类型

➤ 在实际的业务数据处理中，为了更准确地计算结果，需要使用不同精度的数据类型。

| 类型                           | 描述——NumPy基本数据类型与其取值范围（只展示一部分）                                               |
|------------------------------|-----------------------------------------------------------------------------|
| bool                         | 用一位存储的布尔类型（值为TRUE或FALSE）                                                    |
| inti                         | 由所在平台决定其精度的整数（一般为int32或int64）                                               |
| int8, int16, int32, .....    | 整数，范围为-128至127，范围为-2 <sup>31</sup> ~2 <sup>31</sup> -1，范围为-32768至32767， ... |
| uint8, uint16, uint32, ..... | 无符号整数，范围为0至255，范围为0至65535，范围为0至~2 <sup>32</sup> -1， ...                     |
| .....                        |                                                                             |
| float16, float32, .....      | 半精度浮点数，单精度浮点数，双精度浮点数                                                        |
| complex64, complex128        | 复数                                                                          |
| .....                        |                                                                             |

# 创建数组对象

## ➤ 数组数据类型转换

```
In[15]:      print('转换为: ', np.float64(42))  #整型转换为浮点型
```

```
Out[15]:      转换为:  42.0
```

```
In[16]:      print('转换为: ', np.int8(42.0))  #浮点型转换为整型
```

```
Out[16]:      转换为:  42
```

```
In[17]:      print('转换为: ', np.bool(42))  #整型转换为布尔型
```

```
Out[17]:      转换为:  True
```

# 创建数组对象

## ➤ 数组数据类型转换

```
In[18]:      print('转换为: ', np.bool(0))  #整型转换为布尔型
```

```
Out[18]:     转换为:  False
```

```
In[19]:      print('转换为: ', np.float(True))  #布尔型转换为浮点型
```

```
Out[19]:     转换为:  1.0
```

```
In[20]:      print('转换为: ', np.float(False))  #布尔型转换为浮点型
```

```
Out[20]:     转换为:  0.0
```

# 创建数组对象

## ➤ 创建数据类型

- 创建一个存储餐饮企业库存信息的数据类型。其中，用一个长度为40个字符的字符串来记录商品的名称，用一个64位的整数来记录商品的库存数量，最后用一个64位的双精度浮点数来记录商品的价格，具体步骤如下：

```
In[21]: df = np.dtype([("name", np.str_, 40), ("numitems", np.int64), ("price", np.float64)])  
        print('数据类型为：', df)
```

```
Out[21]: 数据类型为： [('name', '<U40'), ('numitems', '<i8'), ('price', '<f8')]
```

- dtype也可以作为参数创建特定的类型的数组

# 创建数组对象

## ➤ 创建数据类型

- 查看数据类型，可以直接查看或者使用numpy.dtype函数查看。

```
In[22]:      print('数据类型为： ', df["name"])
```

```
Out[22]:      数据类型为： <U40
```

```
In[23]:      print('数据类型为： ', np.dtype(df["name"]))
```

```
Out[23]:      数据类型为： <U40
```

# 创建数组对象

## ➤ 创建数据类型

- 在使用array函数创建数组时，数组的数据类型默认是浮点型。自定义数组数据，则可以预先指定数据类型

```
In[24]: itemz = np.array([("tomatoes", 42, 4.14), ("cabbages", 13, 1.72)], dtype=df)
        print('自定义数据为: ', itemz)
```

```
Out[24]: 自定义数据为: [('tomatoes', 42, 4.14) ('cabbages', 13, 1.72)]
```

```
df = np.dtype([("name", np.str_, 40), ("numitems", np.int64), ("price", np.float64)])
```



# 生成随机数

---

- 手动创建数组往往达不到实验要求，Numpy提供了强大的生成随机数的功能。
- 然而，真正的随机数很难获得，实际中使用的都是伪随机数。
- 大部分情况下，伪随机数就能满足需求。当然，某些特殊情况例外(比如进行高精度的模拟实验)
- 对于Numpy，与随机数相关的函数都在**random模块**中，其中包括了可以生成服从多种概率分布随机数的函数。
- 下面介绍一些常用的随机数生成方法。

# 生成随机数

```
#!/usr/bin/python
```

```
# -*- coding: UTF-8 -*-
```

```
import random
```

#Python内建模块，生成的是list，或者单个随机数

```
print( random.randint(1,10) )
```

# 产生 1 到 10 的一个整数型随机数

```
print( random.random() )
```

# 产生 0 到 1 之间的随机浮点数

```
print( random.uniform(1.1,5.4) )
```

# 产生 1.1 到 5.4 之间的随机浮点数，区间可以不是整数

```
print( random.choice('tomorrow') )
```

# 从序列中随机选取一个元素

```
print( random.randrange(1,100,2) )
```

# 生成从1到100的间隔为2的随机整数

```
a=[1,3,5,6,7]
```

# 将序列a中的元素顺序打乱

```
random.shuffle(a)
```

```
print(a)
```

# 生成随机数

## ➤ 无约束条件下生成随机数

- 生成100个0-1之间的随机数，即含有100个随机元素的一维数组

```
In[25]: print('生成的随机数组为：\n', np.random.random(100))
```

```
Out[25]: 生成的随机数组为：  
[0.75946624 0.32403519 0.45156435 0.15301262 0.5466775  0.84605046  
0.17080126 0.69076568 0.90944317 0.92458982 0.64758714 0.86632062  
0.64711029 0.79839372 0.25877929 0.50130451 0.05525751 0.57771451  
0.85348644 0.60981466 0.13559104 0.86325133 0.85926575 0.35266513  
0.73434223 0.55986394 0.09632885 0.33584189 0.50353657 0.47070543  
0.83600292 0.72432525 0.5941884  0.52633137 0.10449681 0.05879993  
0.56039778 0.38883987 0.31812864 0.07824741 0.50415019 0.90873665  
0.76296255 0.09991549 0.05827797 0.96862287 0.75540815 0.1081329  
0.49204804 0.68595159 0.4948226  0.42642197 0.20368589 0.64483586  
0.20556043 0.17892559 0.30848999 0.66396742 0.99889869 0.97470543  
0.96726173 0.20558211 0.17261609 0.86407114 0.62561625 0.49406964  
0.10720964 0.49829874 0.0886082  0.6681555  0.55979343 0.49613836
```

# 生成随机数

- 生成服从均匀分布的随机数
- 生成10\*5的二维数组，矩阵元素符合均匀分布

```
In[26]: print('生成的随机数组为: \n', np.random.rand(10, 5))
```

```
Out[26]: 生成的随机数组为:  
[[0.49003214 0.95476764 0.80478427 0.02563938 0.73185181]  
 [0.83066547 0.86510738 0.33880589 0.81808229 0.23000459]  
 [0.28728246 0.12556771 0.9790872  0.79213652 0.94980362]  
 [0.83897353 0.18182662 0.90904378 0.00172149 0.54878672]  
 [0.50801147 0.67028615 0.92488998 0.71749045 0.72056861]  
 [0.61933732 0.53879096 0.56671656 0.4520065  0.69900323]  
 [0.90862367 0.13648924 0.55345897 0.77281637 0.26118587]  
 [0.51397543 0.48149869 0.5896165  0.84167331 0.57252252]  
 [0.02712242 0.53574904 0.02550368 0.33877762 0.19738869]  
 [0.10046505 0.60883213 0.71470222 0.37963987 0.53927313]]
```

# 生成随机数

- 生成服从正态分布的随机数
- 生成10\*5的二维数组，矩阵元素符合正态分布

```
In[27]:      print('生成的随机数组为：\n', np.random.randn(10, 5))
```

```
Out[27]:      生成的随机数组为：
[[ 0.4325805 -0.98168945 -0.5376785  0.53174084 -1.2785329 ]
 [ 1.34082081 -0.25767934  1.722428  0.34583182 -0.45464513]
 [ 0.69941229 -1.28239453 -0.98410001  0.23979938 -0.01257589]
 [ 0.00850335  0.696565  1.5523713  1.54642731 -0.43864744]
 [ 0.00256476 -0.76022471  0.81188766  1.21809275  0.08206837]
 [ 0.22081286 -0.71360927 -0.22065364  0.76769452  0.79808283]
 [ 0.47247358 -1.43741152  0.33592785 -1.56177824 -1.03810669]
 [-0.22089089  1.00881105  0.43125875  0.29081499  1.10204025]
 [-0.91473289  0.82456542 -0.96201081  0.13506441  0.54720484]
 [-0.26405772 -1.36281329  0.90194283 -1.91379344 -1.47086922]]
```

# 生成随机数

- 生成给定上下范围的随机数
- 创建一个最小值不低于 2、最大值不高于 10 的 2 行 5 列数组

```
In[28]:      print('生成的随机数组为: \n', np.random.randint(2, 10, size = [2,5]))
```

```
Out[28]:      生成的随机数组为:  
           [[6 6 6 6 8]  
           [9 6 6 8 4]]
```

# 生成随机数

## random模块常用随机数生成函数

| 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     | 对一个序列进行随机排序。             |
| binomial    | 产生二项分布的随机数。              |
| normal      | 产生正态（高斯）分布的随机数。          |
| beta        | 产生beta分布的随机数。            |
| chisquare   | 产生卡方分布的随机数。              |
| gamma       | 产生gamma分布的随机数。           |
| uniform     | 产生在[0,1)中均匀分布的随机数。       |

# 生成随机数

---

## np.random.uniform的用法

*np.random.uniform(low=0.0, high=1.0, size=None)*

- 作用：可以生成[low,high)中均匀分布的随机数，可以是单个值，也可以是一维数组，也可以是多维数组
- 参数介绍：
  - ▣ low : float型，或者是数组类型的，默认为0
  - ▣ high: float型，或者是数组类型的，默认为1
  - ▣ size: int型，或元组，默认为空



# 生成随机数

## np.random.uniform的用法

```
In[1]: import Numpy as np  
       np.random.uniform()      # 默认为0到1
```

```
Out[1]: 0.827455693512018
```

```
In[2]: np.random.uniform(1, 5)
```

```
Out[2]: 2.93533586182789
```

```
In[3]: np.random.uniform(1, 5, 4)  #生成一维数组
```

```
Out[3]: array([ 3.18487512, 1.40233721, 3.17543152, 4.06933042])
```

# 生成随机数

## np.random.uniform的用法

In[1]:        **np.random.uniform**(1, 5, (4,3))        **#生成4\*3的数组**

Out[1]:        array([[ 2.33083328, 1.592934 , 2.38072 ],  
                 [ 1.07485686, 4.93224857, 1.42584919],  
                 [ 3.2667912 , 4.57868281, 1.53218578],  
                 [ 4.17965117, 3.63912616, 2.83516143]])

In[2]:        **np.random.uniform**([1,5], [5,10])

Out[2]:        array([ 2.74315143, 9.4701426 ])

# 通过索引访问数组

## 1. 一维数组的索引

➤ Numpy以提供高效率的数组著称，这主要归功于索引的易用性。

```
In[29]: arr = np.arange(10)
        print('索引结果为: ', arr[5]) #用整数作为下标可以获取数组中的某个元素
```

**arr = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])**

```
Out[29]: 索引结果为: 5
```

```
In[30]: print('索引结果为: ', arr[3:5]) #用范围作为下标获取数组的一个切片，包括arr[3]不包括arr[5]
```

```
Out[30]: 索引结果为: [3 4]
```

```
In[31]: print('索引结果为: ', arr[:5]) #省略开始下标，表示从arr[0]开始
```

```
Out[31]: 索引结果为: [0 1 2 3 4]
```

```
In[32]: print('索引结果为: ', arr[-1]) #下标可以使用负数，-1表示从数组后往前数的第一个元素
```

```
Out[32]: 索引结果为: 9
```

# 通过索引访问数组

## 1. 一维数组的索引

```
arr = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In[33]: arr[2:4] = 100,101  
print('索引结果为: ', arr) #下标还可以用来修改元素的值
```

```
Out[33]: 索引结果为: [ 0  1 100 101  4  5  6  7  8  9]
```

```
In[34]: #范围中的第三个参数表示步长, 2表示隔一个元素取一个元素  
print('索引结果为: ', arr[1:-1:2])
```

```
Out[34]: 索引结果为: [ 1 101  5  7]
```

```
In[35]: print('索引结果为: ', arr[5:1:-2]) #步长为负数时, 开始下标必须大于结束下标
```

```
Out[35]: 索引结果为: [ 5 101]
```

# 通过索引访问数组

## 2. 多维数组的索引

```
In[36]: arr = np.array([[1, 2, 3, 4, 5],[4, 5, 6, 7, 8], [7, 8, 9, 10, 11]])  
        print('创建的二维数组为: \n',arr)
```

```
Out[36]: 创建的二维数组为:  
[[ 1  2  3  4  5]  
 [ 4  5  6  7  8]  
 [ 7  8  9 10 11]]
```

```
In[37]: print('索引结果为: ', arr[0, 3:5]) #索引第0行中第3和4列的元素
```

```
Out[37]: 索引结果为: [4 5]
```

# 通过索引访问数组

## 2. 多维数组的索引

```
In[38]: #索引第2和3行中第3 ~ 5列的元素  
print('索引结果为: \n', arr[1:, 2:])
```

```
Out[38]: 索引结果为:  
[[ 6  7  8]  
 [ 9 10 11]]
```

创建的二维数组为:

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

```
In[39]: print('索引结果为: ', arr[:, 2]) #索引第2列的元素
```

```
Out[39]: 索引结果为: [3 6 9]
```

# 通过索引访问数组

## 2. 多维数组的索引（使用整数和布尔值索引访问数据）

创建的二维数组为：

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

In[40]: `#从两个序列的对应位置取出两个整数来组成下标`  
`print('索引结果为: ', arr[[0,1,2),(1,2,3)])` `#实际取的数是: arr[0,1], arr[1,2], arr[2,3]`

Out[40]: 索引结果为: [ 2 6 10]

In[41]: `print('索引结果为: \n', arr[1:,(0,2,3)])` `#索引第2、3行中第0、2、3列的元素`

Out[41]: 索引结果为:  
[[ 4 6 7]  
 [ 7 9 10]]

In[42]: `mask = np.array([1,0,1],dtype = np.bool)` `#mask是一个布尔数组，它索引第1、3行`  
`print('索引结果为: ', arr[mask,2])`

Out[42]: 索引结果为: [3 9]

# 变换数组的形态

## ➤ 改变数组形状

|          |                                                                                        |                                                                |
|----------|----------------------------------------------------------------------------------------|----------------------------------------------------------------|
| In[43]:  | <code>arr = np.arange(12)      #创建一维数组</code><br><code>print('创建的一维数组为: ', arr)</code> |                                                                |
| Out[43]: | 创建的一维数组为: [ 0  1  2  3  4  5  6  7  8  9 10 11]                                        |                                                                |
| In[44]:  | <code>print('新的一维数组为: \n', arr.reshape(3,4))      #设置数组的形状, 但arr本身没变</code>            |                                                                |
| Out[44]: | 新的一维数组为:<br>[[ 0  1  2  3]<br>[ 4  5  6  7]<br>[ 8  9 10 11]]                          | reshape函数在改变原始数据形状的同时不改变原始数据的值。如果指定的维度和数组的元素数目不吻合, 则函数将出错抛出异常。 |
| In[45]:  | <code>print('数组维度为: ', arr.reshape(3,4).ndim) #查看数组维度</code>                           |                                                                |
| Out[45]: | 数组维度为: 2                                                                               |                                                                |



# 变换数组的形态

## ➤ 使用ravel函数展平数组

|          |                                                                      |
|----------|----------------------------------------------------------------------|
| In[46]:  | <pre>arr = np.arange(12).reshape(3,4) print('创建的二维数组为： ', arr)</pre> |
| Out[46]: | <pre>创建的二维数组为： [[ 0  1  2  3]  [ 4  5  6  7]  [ 8  9 10 11]]</pre>   |
| In[47]:  | <pre>print('数组展平后为： ', arr.ravel())</pre>                            |
| Out[47]: | <pre>数组展平后为：  [ 0  1  2  3  4  5  6  7  8  9 10 11]</pre>            |

# 变换数组的形态

## ➤ 使用flatten函数展平数组

➤ flatten函数也可以完成数组的展平工作

➤ flatten与ravel函数的区别在于，flatten函数可以选择横向或纵向展平

|          |                                                      |                                                                       |
|----------|------------------------------------------------------|-----------------------------------------------------------------------|
| In[48]:  | <code>print('数组展平为: ',arr.flatten()) #横向展平</code>    | <div>创建的二维数组为:<br/>[[ 0 1 2 3]<br/>[ 4 5 6 7]<br/>[ 8 9 10 11]]</div> |
| Out[48]: | 数组展平为: [ 0 1 2 3 4 5 6 7 8 9 10 11]                  |                                                                       |
| In[49]:  | <code>print('数组展平为: ',arr.flatten('F')) #纵向展平</code> |                                                                       |
| Out[49]: | 数组展平为: [ 0 4 8 1 5 9 2 6 10 3 7 11]                  |                                                                       |

# 变换数组的形态

## ➤ 组合数组

➤ Numpy除了可以改变数组“形状”外，还可以对数组进行组合。

➤ 组合主要有横向组合与纵向组合。

□ 使用**hstack**函数实现数组横向组合： `np.hstack((arr1,arr2))`

□ 使用**vstack**函数实现数组纵向组合： `np.vstack((arr1,arr2))`

□ 使用**concatenate**函数实现数组横向组合： `np.concatenate((arr1,arr2),axis = 1))`

□ 使用**concatenate**函数实现数组纵向组合： `np.concatenate((arr1,arr2),axis = 0))`

# 变换数组的形态

## ➤ 首先创建两个二维数组arr1, arr2

```
In[1]: arr1 = np.arange(12).reshape(3,4)
print('创建的数组arr1为: \n',arr1)
```

```
Out[1]: 创建的数组arr1为:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
In[2]: arr2 = arr1*3
print('创建的数组arr2为: \n',arr2)
```

```
Out[2]: 创建的数组arr2为:
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

# 变换数组的形态

➤ 使用hstack函数实现数组横向组合: `np.hstack((arr1,arr2))`

|         |                                                                                                     |               |
|---------|-----------------------------------------------------------------------------------------------------|---------------|
| In[3]:  | <code>print('横向组合为: \n',np.hstack((arr1,arr2)))</code>                                              | #hstack函数横向组合 |
| Out[3]: | <pre>横向组合为: [[ 0  1  2  3  0  3  6  9]  [ 4  5  6  7 12 15 18 21]  [ 8  9 10 11 24 27 30 33]]</pre> |               |

创建的数组arr1为:

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

创建的数组arr2为:

```
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

# 变换数组的形态

- 使用vstack函数实现数组纵向组合: `np.vstack((arr1,arr2))`

In[3]: `print('纵向组合为: \n',np.vstack((arr1,arr2))) #vstack函数纵向组合`

Out[3]: 纵向组合为:  
[[ 0 1 2 3]  
[ 4 5 6 7]  
[ 8 9 10 11]  
[ 0 3 6 9]  
[12 15 18 21]  
[24 27 30 33]]

创建的数组arr1为:  
[[ 0 1 2 3]  
[ 4 5 6 7]  
[ 8 9 10 11]]

创建的数组arr2为:  
[[ 0 3 6 9]  
[12 15 18 21]  
[24 27 30 33]]

# 变换数组的形态

- 使用concatenate函数实现数组横向组合: `np.concatenate((arr1,arr2),axis = 1)`

|         |                                                                                                     |
|---------|-----------------------------------------------------------------------------------------------------|
| In[3]:  | <code>print('横向组合为: \n',np.concatenate((arr1,arr2),axis = 1)) #concatenate函数横向组合</code>             |
| Out[3]: | <pre>横向组合为: [[ 0  1  2  3  0  3  6  9]  [ 4  5  6  7 12 15 18 21]  [ 8  9 10 11 24 27 30 33]]</pre> |

创建的数组arr1为:

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

创建的数组arr2为:

```
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]
```

# 变换数组的形态

- 使用concatenate函数实现数组纵向组合: `np.concatenate((arr1,arr2),axis = 0)`

|         |                                                                                                                |
|---------|----------------------------------------------------------------------------------------------------------------|
| In[3]:  | <code>print('纵向组合为: ',np.concatenate((arr1,arr2),axis = 0)) #concatenate函数纵向组合</code>                          |
| Out[3]: | 纵向组合为:<br>[[ 0  1  2  3]<br>[ 4  5  6  7]<br>[ 8  9 10 11]<br>[ 0  3  6  9]<br>[12 15 18 21]<br>[24 27 30 33]] |

创建的数组arr1为:  
[[ 0 1 2 3]  
[ 4 5 6 7]  
[ 8 9 10 11]]

创建的数组arr2为:  
[[ 0 3 6 9]  
[12 15 18 21]  
[24 27 30 33]]



# 变换数组的形态

## ➤ 切割数组

- Numpy除了可以对数组进行组合以外，还可以对数组进行切割。
- 可以将数组切割成相同大小的子数组，也可以指定原数组中需要分割的位置。
  - 使用hsplit函数实现数组横向分割： `np.hsplit(arr1, 2)`
  - 使用vsplit函数实现数组纵向分割： `np.vsplit(arr, 2)`
  - 使用split函数实现数组横向分割： `np.split(arr, 2, axis=1)`
  - 使用split函数实现数组纵向分割： `np.split(arr, 2, axis=0)`

# 变换数组的形态

- 首先创建一个二维数组arr

|         |                                                                                   |
|---------|-----------------------------------------------------------------------------------|
| In[1]:  | <pre>arr = np.arange(16).reshape(4,4) print('创建的二维数组为：\n',arr)</pre>              |
| Out[1]: | <pre>创建的二维数组为： [[ 0  1  2  3]  [ 4  5  6  7]  [ 8  9 10 11]  [12 13 14 15]]</pre> |

# 变换数组的形态

- 使用hsplit函数实现数组横向分割: `np.hsplit(arr, 2)`

In[2]: `print('横向分割为: \n',np.hsplit(arr, 2)) #hsplit函数横向分割`

Out[2]: 横向分割为:  
[array([[ 0, 1],  
[ 4, 5],  
[ 8, 9],  
[12, 13]]), array([[ 2, 3],  
[ 6, 7],  
[10, 11],  
[14, 15]])]

创建的二维数组为:

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

# 变换数组的形态

- 使用vsplit函数实现数组纵向分割: `np.vsplit(arr, 2)`

In[2]: `print('纵向分割为: \n', np.vsplit(arr, 2))` #vsplit函数纵向分割

Out[2]: 纵向分割为:  
[array([[0, 1, 2, 3],  
[4, 5, 6, 7]]), array([[ 8, 9, 10, 11],  
[12, 13, 14, 15]])]

创建的二维数组为:

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

# 变换数组的形态

- 使用split函数实现数组横向分割: *np.split(arr, 2, axis=1)*

|         |                                                                                                                                               |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| In[2]:  | <code>print('横向分割为: \n',np.split(arr, 2, axis=1)) #split函数横向分割</code>                                                                         |
| Out[2]: | <code>横向分割为:<br/>[array([[ 0, 1],<br/>[ 4, 5],<br/>[ 8, 9],<br/>[12, 13]]), array([[ 2, 3],<br/>[ 6, 7],<br/>[10, 11],<br/>[14, 15]])]</code> |

# 变换数组的形态

- 使用split函数实现数组纵向分割: *np.split(arr, 2, axis=0)*

|         |                                                                                                                   |
|---------|-------------------------------------------------------------------------------------------------------------------|
| In[2]:  | <code>print('纵向分割为: \n',np.split(arr, 2, axis=0)) #split函数纵向分割</code>                                             |
| Out[2]: | <code>纵向分割为:<br/>[array([[0, 1, 2, 3],<br/>[4, 5, 6, 7]]), array([[ 8, 9, 10, 11],<br/>[12, 13, 14, 15]])]</code> |

# 目录

---



# 掌握 NumPy 矩阵与通用函数

---

## NumPy矩阵

- NumPy对于多维数组的运算，默认情况下并不进行矩阵运算。若想对数组进行矩阵运算，需要使用另外的函数。
- 在NumPy中，矩阵是ndarray的子类。数组和矩阵有着重要的区别。
- NumPy提供了两个基本的对象：一个N维数组对象和一个通用函数对象。其他对象都是在它们之上构建的。
- 矩阵是继承自NumPy数组对象的二维数组对象。与数学概念中的矩阵一样，NumPy中的矩阵也是二维的。



# 掌握 NumPy 矩阵与通用函数

## NumPy中数组array和矩阵matrix区别

1. array可以是1维, 二维, ....., N维的, 而matrix只是二维的

➤ 具体矩阵生成方式的不同:

```
import numpy as np
```

```
a1 = np.array([[1, 2], [3, 4]])
```

```
b1 = np.mat([[1, 2], [3, 4]])
```

```
a2 = np.array(( [1, 2], [3, 4] ))
```

```
b2 = np.mat(( [1, 2], [3, 4] ))
```

```
a3 = np.array(((1,2), (3,4)))
```

```
b3 = np.mat(((1,2), (3,4)))
```

```
b4 = np.mat('1 2; 3 4')
```

```
print("\n",a1,"\n",b1,"\n",a2,"\n",b2,"\n",a3,"\n",b3,"\n",b4)
```

- 这些函数变化的无非就是把函数大括号内的"[]"换成"()", 将括起来的认为一个整体。
- 不同之处在于 b4 内用引号、空格和分号来产生矩阵, 这个方法只可以在 matrix() 函数中使用, 不可以写成的 a4 = np.array('1 2; 3 4') 。

输出结果均为:

```
[[1 2]  
[3 4]]
```

# 掌握 NumPy 矩阵与通用函数

## Numpy中数组array和矩阵matrix区别

### 2. 矩阵性质的差异

- matrix()和 array ()函数都可以通过对象后面加上 **.T** 得到其转置。
- 但是matrix()还可以在后面加 **.H** 得到共轭矩阵, 加 **.I** 得到逆矩阵, array()就不可以。例子如下:

#### 1) 转置:

```
import numpy as np
```

```
a1 = np.array([[1, 2], [3, 4]])
```

```
b1 = np.mat([[1, 2], [3, 4]])
```

```
print(a1.T)
```

```
print(b1.T)
```

输出结果均为:

```
[[1 3]  
 [2 4]]
```

# 掌握 NumPy 矩阵与通用函数

## Numpy中数组array和矩阵matrix区别

### 2. 矩阵性质的差异

- matrix()和 array ()函数都可以通过对象后面加上 **.T** 得到其转置。
- 但是matrix()还可以在后面加 **.H** 得到共轭矩阵, 加 **.I** 得到逆矩阵, array()就不可以。例子如下:

### 2) 共轭和逆矩阵

```
import numpy as np  
a1 = np.array([[1, 2], [3, 4]])  
print(a1.H)
```

结果为:

AttributeError: 'numpy.ndarray' object has no attribute 'H'

程序最后一行换成: `print(a1.I)`

结果为:

AttributeError: 'numpy.ndarray' object has no attribute 'I'

# 掌握 NumPy 矩阵与通用函数

## Numpy中数组array和矩阵matrix区别

### 2. 矩阵性质的差异

- 所以，`array()` 就不具有 `.H` `.I`，但是，如果用 `b1` 计算共轭矩阵和逆矩阵则是可以的，如下：

```
import numpy as np
```

```
b1 = np.mat([[1, 2], [3, 4]])
```

```
print(b1.H)
```

```
print(b1.I)
```

结果为（共轭矩阵和逆矩阵分别为）：

```
[[1 3]
 [2 4]]
[[-2.  1.]
 [ 1.5 -0.5]]
```

# 掌握 NumPy 矩阵与通用函数

## Numpy中数组array和矩阵matrix区别

### 3. 在矩阵乘法的不同体现

- 观察以下两个输出语句结果的不同：

```
import numpy as np
```

```
a1 = np.array([[1, 2], [3, 4]])
```

```
c1 = np.array([[5, 6], [7, 8]])
```

```
b1 = np.mat([[1, 2], [3, 4]])
```

```
d1 = np.mat([[5, 6], [7, 8]])
```

```
print("a1乘c1的结果: ",a1*c1)
```

```
print("b1乘d1的结果: ",b1*d1)
```

- 可以得知：

- array()函数的乘法是矩阵元素所对应位置的两个数进行相乘！

- 而mat()函数是遵循矩阵乘法规则。

- 所以一定要谨慎使用这两个函数。

**输出结果会有不同，分别为：**

**a1乘c1的结果：**  $\begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$

**b1乘d1的结果：**  $\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$

# 掌握 NumPy 矩阵与通用函数

## Numpy中数组array和矩阵matrix区别

### 3. 在矩阵乘法的不同体现

- 不过，无论用array()函数还是mat()函数，若让他们都遵循矩阵乘法的规则，可以对此二阶矩阵应用dot()函数，看下面计算：

```
print(dot(a1,c1))
```

```
print(dot(b1,d1))
```

输出结果均为：

```
[[19 22]
```

```
[43 50]]
```

# 掌握 NumPy 矩阵与通用函数

## Numpy中数组array和矩阵matrix区别

### 3. 在矩阵乘法的不同体现

- 根据上文分析，如果我们要算一个矩阵的平方（即两个相同矩阵相乘）。把输出函数变一下，观察一下结果的不同：

```
print("a1的平方",a1**2)
```

```
print("b1的平方",b1**2)
```

**输出结果为：**

**a1的平方**  $\begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$

**b1的平方**  $\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$

- 结果正好验证了之前的结论。用array()函数求的平方是对应位置之积形成的矩阵。用mat()函数求的平方实际上是遵循矩阵计算得出来的

# 掌握 NumPy 矩阵与通用函数

---

## NumPy中数组array和矩阵matrix区别

➤ 总结一下：

- array()函数的相乘中：**\***代表点乘（对应元素相乘），**dot()**代表矩阵乘积。
- mat()函数的乘法中：**\***代表矩阵乘，**multiply()**代表点乘



# 掌握 NumPy 矩阵与通用函数

---

## 创建NumPy矩阵

- 本小节将讲解使用`mat`、`matrix` 以及`bmata` (block matrix) 函数来创建矩阵。
- 使用`mat`函数创建矩阵时，若输入`matrix`或`ndarray`对象，则不会为它们创建副本，因此，调用`mat`函数和调用`matrix (data, copy = False)` 等价。

# 创建NumPy矩阵

---

## 创建与组合矩阵

- 使用mat函数创建矩阵: `matr1 = np.mat("1 2 3; 4 5 6; 7 8 9")` #使用分号隔开数据
- 使用matrix函数创建矩阵: `matr2 = np.matrix([[123],[456],[789]])`
- 使用bmat函数合成矩阵: `np.bmat("arr1 arr2; arr1 arr2")`

# 创建NumPy矩阵

## ➤ 使用mat函数创建矩阵

|        |                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------|
| In[1]: | <pre>import numpy as np          #导入NumPy库 matr1 = np.mat("1 2 3;4 5 6;7 8 9") #使用分号隔开数据 print('创建的矩阵为：\n',matr1)</pre> |
|--------|-------------------------------------------------------------------------------------------------------------------------|

|         |                                                |
|---------|------------------------------------------------|
| Out[1]: | <pre>创建的矩阵为： [[1 2 3]  [4 5 6]  [7 8 9]]</pre> |
|---------|------------------------------------------------|

|        |                        |
|--------|------------------------|
| In[2]: | <pre>matr1.shape</pre> |
|--------|------------------------|

|         |                   |
|---------|-------------------|
| Out[2]: | <pre>(3, 3)</pre> |
|---------|-------------------|

# 创建NumPy矩阵

## ➤ 使用matrix函数创建矩阵

|         |                                                                                                                        |
|---------|------------------------------------------------------------------------------------------------------------------------|
| In[1]:  | <pre>import numpy as np          #导入NumPy库 matr2 = np.matrix([[1,2,3],[4,5,6],[7,8,9]]) print('创建的矩阵为：\n',matr2)</pre> |
| Out[1]: | <pre>创建的矩阵为： [[1 2 3]  [4 5 6]  [7 8 9]]</pre>                                                                         |
| In[2]:  | <pre>matr2.shape</pre>                                                                                                 |
| Out[2]: | <pre>(3, 3)</pre>                                                                                                      |

# 创建NumPy矩阵

➤ 使用bmat函数合成矩阵: `np.bmat("arr1 arr2; arr1 arr2")`

```
In[1]: import numpy as np
      arr1 = np.eye(3)
      print('创建的数组1为: \n',arr1)
```

```
Out[1]: 创建的数组1为:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
In[2]: arr2 = 3*arr1
      print('创建的数组2为: \n',arr2)
```

```
Out[2]: 创建的数组2为:
[[3. 0. 0.]
 [0. 3. 0.]
 [0. 0. 3.]]
```

```
In[3]: print('创建的矩阵为: \n', np.bmat("arr1
      arr2; arr1 arr2"))
```

```
Out[3]: 创建的矩阵为:
[[1. 0. 0. 3. 0. 0.]
 [0. 1. 0. 0. 3. 0.]
 [0. 0. 1. 0. 0. 3.]
 [1. 0. 0. 3. 0. 0.]
 [0. 1. 0. 0. 3. 0.]
 [0. 0. 1. 0. 0. 3.]]
```

# 创建NumPy矩阵

---

## 矩阵的运算

- 矩阵与数相乘: `matr1*3`
- 矩阵相加减: `matr1±matr2`
- 矩阵相乘: `matr1*matr2`
- 矩阵对应元素相乘: `np.multiply(matr1,matr2)`
  
- 在NumPy中, 矩阵计算是针对整个矩阵中的每个元素进行的, 与使用for循环相比, 其在运算速度上更快

# 创建NumPy矩阵

## 矩阵运算

|         |                                                                                                  |
|---------|--------------------------------------------------------------------------------------------------|
| In[1]:  | <pre>import numpy as np matr1 = np.mat("1 2 3;4 5 6;7 8 9") #创建矩阵 print('创建的矩阵为：\n',matr1)</pre> |
| Out[2]: | <pre>创建的矩阵为： [[1 2 3]  [4 5 6]  [7 8 9]]</pre>                                                   |
| In[3]:  | <pre>matr2 = matr1*3 #矩阵与数相乘 print('创建的矩阵为：\n',matr2)</pre>                                      |
| Out[3]: | <pre>创建的矩阵为： [[ 3  6  9]  [12 15 18]  [21 24 27]]</pre>                                          |

# 创建NumPy矩阵

创建的matr1为:

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

创建的matr2为:

```
[[ 3  6  9]  
[12 15 18]  
[21 24 27]]
```

## 矩阵运算

```
In[1]: print('矩阵相加结果为: \n',matr1+matr2) #矩阵相加
```

```
Out[1]: 矩阵相加结果为:  
[[ 4  8 12]  
[16 20 24]  
[28 32 36]]
```

```
In[2]: print('矩阵相减结果为: \n',matr1-matr2) #矩阵相减
```

```
Out[2]: 矩阵相减结果为:  
[[ -2  -4  -6]  
[ -8 -10 -12]  
[-14 -16 -18]]
```



# 创建NumPy矩阵

创建的matr1为:

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

创建的matr2为:

```
[[ 3  6  9]  
[12 15 18]  
[21 24 27]]
```

## 矩阵运算

In[1]: `print('矩阵相乘结果为: \n',matr1*matr2) #矩阵相乘`

Out[1]: 矩阵相乘结果为:  
[[ 90 108 126]  
[198 243 288]  
[306 378 450]]

In[2]: `print('矩阵对应元素相乘结果为: \n',np.multiply(matr1,matr2)) #矩阵对应元素相乘`

Out[2]: 矩阵对应元素相乘结果为:  
[[ 3 12 27]  
[ 48 75 108]  
[147 192 243]]

# 创建NumPy矩阵

## 矩阵的运算

➤ 矩阵特有属性：

| 属性 | 说明               |
|----|------------------|
| T  | 返回自身的转置          |
| H  | 返回自身的共轭转置        |
| I  | 返回自身的逆矩阵         |
| A  | 返回自身数据的2维数组的一个视图 |

# 创建NumPy矩阵

创建的matr1为:

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

## 矩阵特有属性

```
In[2]: print('矩阵转置结果为: \n',matr1.T) #转置
```

```
Out[2]: 矩阵转置结果为:  
[[1 4 7]  
[2 5 8]  
[3 6 9]]
```

```
In[3]: print('矩阵共轭转置结果为: \n',matr1.H) #共轭转置 (实数的共轭就是其本身)
```

```
Out[3]: 矩阵共轭转置结果为:  
[[1 4 7]  
[2 5 8]  
[3 6 9]]
```

# 创建NumPy矩阵

创建的matr1为:  
[[1 2 3]  
[4 5 6]  
[7 8 9]]

## 矩阵特有属性

```
In[2]: print('矩阵的逆矩阵结果为: \n',matr1.I) #逆矩阵
```

```
Out[2]: 矩阵的逆矩阵结果为:  
[[ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]  
 [-6.30503948e+15  1.26100790e+16 -6.30503948e+15]  
 [ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]]
```

```
In[3]: print('矩阵的二维数组结果为: \n',matr1.A) #返回二维数组的视图
```

```
Out[3]: 矩阵的二维数组结果为:  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

# 认识ufunc函数

---

## 通用函数 (universal function)

- 是一种能够对数组中所有元素进行操作的函数 (元素级运算)
- Ufunc函数是针对ndarray数组进行操作的, 并且都以NumPy数组作为输出
- 因此不需要对数组的每一个元素都进行操作。
- 对一个数组进行重复运算时, **使用ufunc函数比使用math库中的函数效率要高很多。**

# 认识ufunc函数

## 常用的ufunc函数运算有四则运算，比较运算和逻辑运算

- **四则运算：**加 (+)、减 (-)、乘 (\*)、除 (/)、幂 (\*\*)
  - 数组间的四则运算表示对每个数组中的元素分别进行四则运算，所以形状必须相同
- **比较运算：**>、<、==、>=、<=、!=
  - 比较运算返回的结果是一个布尔数组，每个元素为每个数组对应元素的比较结果
- **逻辑运算：**np.any函数表示逻辑 “or” ， np.all函数表示逻辑 “and”
  - 运算结果返回布尔值

# 认识ufunc函数

## ➤ ufunc函数运算——四则运算

|         |                                                                                                      |
|---------|------------------------------------------------------------------------------------------------------|
| In[1]:  | <pre>x = np.array([1,2,3])<br/>y = np.array([4,5,6])<br/><br/>print('数组相加结果为: ', x + y)  #数组相加</pre> |
| Out[1]: | 数组相加结果为: [5 7 9]                                                                                     |
| In[2]:  | <pre>print('数组相减结果为: ', x - y)  #数组相减</pre>                                                          |
| Out[2]: | 数组相减结果为: [-3 -3 -3]                                                                                  |

➤ 矩阵每行中的元素由逗号分隔，二维数组每行中的元素由空格分隔

# 认识ufunc函数

## ➤ ufunc函数运算——四则运算

```
In[4]: print('数组相乘结果为: ', x * y)  #数组相乘
```

```
Out[4]: 数组相乘结果为: [ 4 10 18]
```

```
In[5]: print('数组相除结果为: ', x / y)  #数组相除
```

```
Out[5]: 数组相除结果为: [0.25 0.4  0.5 ]
```

```
In[6]: print('数组幂运算结果为: ', x ** y)  #数组幂运算
```

```
Out[6]: 数组幂运算结果为: [ 1 32 729]
```

```
x = [1,2,3]  
y = [4,5,6]
```



# 认识ufunc函数

## ➤ ufunc函数运算——比较运算

```
In[1]: x = np.array([1,3,5])  
y = np.array([2,3,4])  
print('数组比较结果为: ', x < y)
```

```
Out[1]: 数组比较结果为: [ True False False]
```

```
In[2]: print('数组比较结果为: ', x > y)
```

```
Out[2]: 数组比较结果为: [False False  True]
```

```
In[3]: print('数组比较结果为: ', x == y)
```

```
Out[3]: 数组比较结果为: [False  True False]
```

# 认识ufunc函数

## ➤ ufunc函数运算——比较运算

```
In[4]: print('数组比较结果为: ', x >= y)
```

```
Out[4]: 数组比较结果为: [False True True]
```

```
In[5]: print('数组比较结果为: ', x <= y)
```

```
Out[5]: 数组比较结果为: [ True True False]
```

```
In[6]: print('数组比较结果为: ', x != y)
```

```
Out[6]: 数组比较结果为: [ True False True]
```

# 认识ufunc函数

## ➤ ufunc函数运算——逻辑运算

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

```
Out[2]: array([False,  True, False])
```

```
In[3]: print('数组逻辑运算结果为: ', np.all(x == y))  #np.all()表示逻辑and
```

```
Out[3]: 数组逻辑运算结果为:  False
```

```
In[4]: print('数组逻辑运算结果为: ', np.any(x == y))  #np.any()表示逻辑or
```

```
Out[4]: 数组逻辑运算结果为:  True
```

# 认识ufunc函数

## ufunc函数的广播机制

➤ 广播 (broadcasting) 是指不同形状的数组之间执行算术运算的方式

需要遵循4个原则：

- 让所有输入数组都向其中shape最长的数组看齐，shape中不足的部分都通过在前面加1补齐
- 输出数组的shape是输入数组shape的各个轴上的最大值
- 如果输入数组的某个轴和输出数组的对应轴的长度相同或者其长度为1时，这个数组能够用来计算，否则出错
- 当输入数组的某个轴的长度为1时，沿着此轴运算时都用此轴上的第一组值

# 认识ufunc函数

## ufunc函数的广播机制

### ➤ 一维数组的广播机制

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + [1 \ 2 \ 3] \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix}$$

# 认识ufunc函数

## ufunc函数的广播机制：一维数组的广播

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + [1 \ 2 \ 3] \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix}$$

```
In[1]: arr1 = np.array([[0,0,0],[1,1,1],[2,2,2],[3,3,3]])
       print('创建的数组1为: \n', arr1)

Out[1]: 创建的数组1为:
        [[0 0 0]
         [1 1 1]
         [2 2 2]
         [3 3 3]]

In[2]: print('数组1的shape为: ', arr1.shape)

Out[2]: 数组1的shape为: (4, 3)
```

```
In[3]: arr2 = np.array([1,2,3])
       print('创建的数组2为: ', arr2)

Out[3]: 创建的数组2为: [1 2 3]

In[4]: print('数组2的shape为: ', arr2.shape)

Out[4]: 数组2的shape为: (3,)

In[5]: print('数组相加结果为: \n', arr1+arr2)

Out[5]: 数组相加结果为:
        [[1 2 3]
         [2 3 4]
         [3 4 5]
         [4 5 6]]
```

# 认识ufunc函数

## ufunc函数的广播机制

### ➤ 二维数组的广播机制

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 3 & 3 & 3 \\ 5 & 5 & 5 \\ 7 & 7 & 7 \end{bmatrix}$$

-- ➡

# 认识ufunc函数

## ➤ ufunc函数的广播机制：二维数组的广播

[0 0 0]

[1 1 1]

[2 2 2]

[3 3 3]

[1]

[2]

[3]

[4]

+

[0 0 0]

[1 1 1]

[2 2 2]

[3 3 3]

[1 1 1]

[2 2 2]

[3 3 3]

[4 4 4]

=

[1 1 1]

[3 3 3]

[5 5 5]

[7 7 7]

|         |                                                                                                            |
|---------|------------------------------------------------------------------------------------------------------------|
| In[1]:  | <code>arr1 = np.array([[0,0,0],[1,1,1],[2,2,2],[3,3,3]])</code><br><code>print('创建的数组1为: \n', arr1)</code> |
| Out[1]: | 创建的数组1为:<br>[[0 0 0]<br>[1 1 1]<br>[2 2 2]<br>[3 3 3]]                                                     |
| In[2]:  | <code>print('数组1的shape为: ', arr1.shape)</code>                                                             |
| Out[2]: | 数组1的shape为: (4, 3)                                                                                         |

|         |                                                                                                   |
|---------|---------------------------------------------------------------------------------------------------|
| In[3]:  | <code>arr2 = np.array([1,2,3,4]).reshape((4,1))</code><br><code>print('创建的数组2为: \n', arr2)</code> |
| Out[3]: | 创建的数组2为:<br>[[1]<br>[2]<br>[3]<br>[4]]                                                            |
| In[4]:  | <code>print('数组2的shape为: ', arr2.shape)</code>                                                    |
| Out[4]: | 数组2的shape为: (4,1)                                                                                 |
| In[5]:  | <code>print('数组相加结果为: \n', <b>arr1+arr2</b>)</code>                                               |
| Out[5]: | 数组相加结果为:<br>[[1 1 1]<br>[3 3 3]<br>[5 5 5]<br>[7 7 7]]                                            |



# 目录

---



# 读写文件

NumPy文件读写主要有二进制的文件读写和文件列表形式的数据读写两种形式

学会读写文件是利用NumPy进行数据处理的基础

- save函数是以二进制的格式**保存数据**: `np.save("../tmp/save_arr", arr)`
- load函数是从二进制的文件中**读取数据**: `np.load("../tmp/save_arr.npy")`
- savez函数可以将多个数组保存到一个文件中: `np.savez('../tmp/savez_arr',arr1,arr2)`
- 存储时可以省略扩展名, 但读取时不能省略扩展名。

# 读写文件

➤ save函数是以二进制的格式保存数据: `np.save("../tmp/save_arr",arr)`

```
In[1]: import numpy as np  #导入NumPy库  
arr = np.arange(100).reshape(10,10)  #创建一个数组  
arr
```

```
Out[1]: 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, 32, 33, 34, 35, 36, 37, 38, 39],  
           [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],  
           [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],  
           [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],  
           [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],  
           [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],  
           [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
In[2]: np.save("../tmp/save_arr",arr) #保存数组
```

# 读写文件

➤ savez函数可以将多个数组保存到一个文件中: *np.savez('../tmp/savez\_arr',arr1,arr2)*

|         |                                                                                                         |           |
|---------|---------------------------------------------------------------------------------------------------------|-----------|
| In[1]:  | <pre>import numpy as np<br/>arr1 = np.array([[1,2,3],[4,5,6]])<br/>arr2 = np.arange(0,1.0,0.1)</pre>    | #导入NumPy库 |
| In[2]:  | <pre>print('保存的数组1为: \n',arr1)<br/>print('保存的数组2为: \n',arr2)</pre>                                      |           |
| Out[2]: | <pre>保存的数组1为:<br/>[[1 2 3]<br/>[4 5 6]]<br/>保存的数组2为:<br/>[0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]</pre> |           |
| In[3]:  | <pre>np.savez('../tmp/savez_arr', arr1, arr2)</pre>                                                     |           |

# 读写文件

➤ load函数是从二进制的文件中读取数据; *np.load("../tmp/save\_arr.npy")*

```
In[1]: import numpy as np                                #导入NumPy库  
loaded_data = np.load("../tmp/save_arr.npy") #读取含有单个数组的文件  
print('读取的数组为: \n', loaded_data)
```

```
Out[1]: 读取的数组为:  
[[ 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 48 49]  
 [50 51 52 53 54 55 56 57 58 59]  
 [60 61 62 63 64 65 66 67 68 69]  
 [70 71 72 73 74 75 76 77 78 79]  
 [80 81 82 83 84 85 86 87 88 89]  
 [90 91 92 93 94 95 96 97 98 99]]
```

# 读写文件

➤ load函数是从二进制的文件中读取数据: `np.load("../tmp/save_arr.npy")`

|         |                                                                                                                                                                                                |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| In[1]:  | <pre>import numpy as np  #导入NumPy库 loaded_data1 = np.load("../tmp/savez_arr.npz")  #读取含有多个数组的文件  print('读取的数组1为: \n', loaded_data1['arr_0']) print('读取的数组2为: \n', loaded_data1['arr_1'])</pre> |
| Out[1]: | <pre>读取的数组1为: [[1 2 3]  [4 5 6]]  读取的数组2为: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]</pre>                                                                                                     |

# 读写文件

## 读取文本格式的数据

在实际的数据分析任务中，更多地使用文本格式的数据，如TXT或CSV格式文件，因此经常使用savetxt, loadtxt, genfromtxt等函数执行对文本格式数据的读取任务

- savetxt函数是将数组写到某种分隔符隔开的文本文件中  
*`np.savetxt("../tmp/arr.txt", arr, fmt="%d", delimiter=",")`*
- loadtxt函数执行的是把文件加载到一个二维数组中  
*`np.loadtxt("../tmp/arr.txt", delimiter=",")`*
- genfromtxt函数面向的是结构化数组和缺失数据  
*`np.genfromtxt("../tmp/arr.txt", delimiter = ",")`*

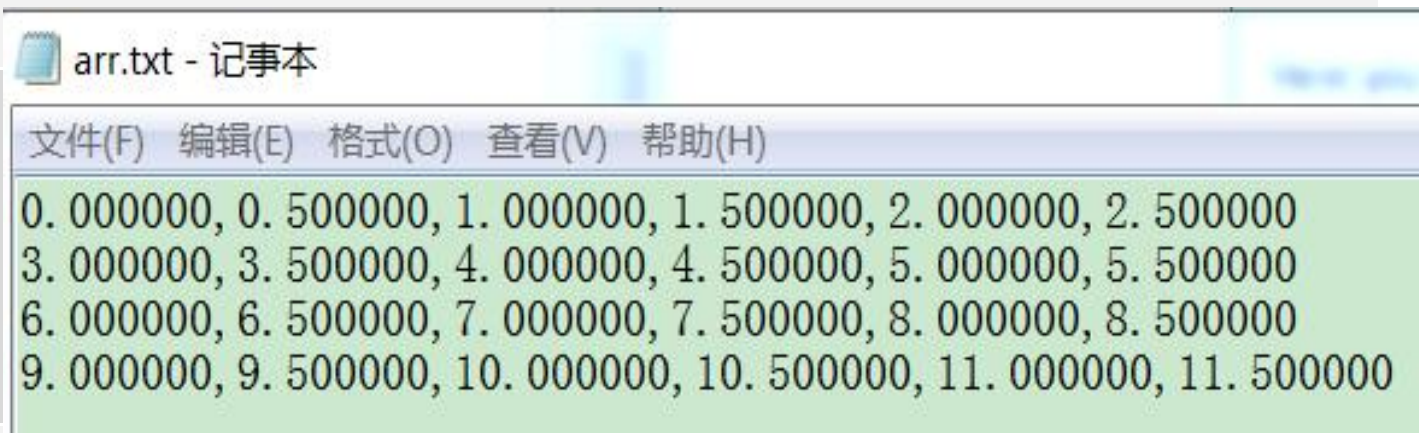
# 读写文件

➤ **savetxt**函数是将数组写到某种分隔符隔开的文本文件中

```
In[1]: import numpy as np  #导入NumPy库  
arr = np.arange(0,12,0.5).reshape(4,-1)  
print('创建的数组为: \n',arr)
```

Numpy会根据剩下的维度计算出数组的另外一个shape属性值

```
Out[1]: 创建的数组为:  
[[ 0.  0.5  1.  1.5  2.  2.5]  
 [ 3.  3.5  4.  4.5  5.  5.5]  
 [ 6.  6.5  7.  7.5  8.  8.5]  
 [ 9.  9.5 10. 10.5 11. 11.5]]
```



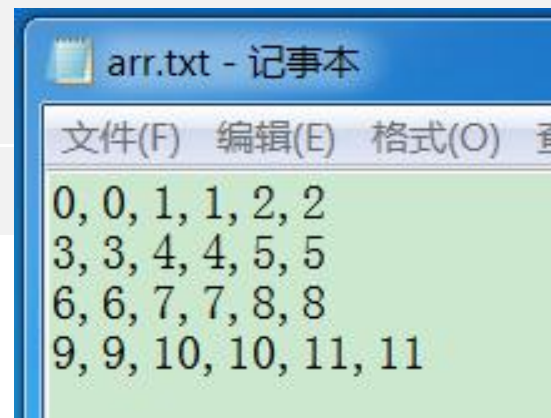
arr.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

0.000000, 0.500000, 1.000000, 1.500000, 2.000000, 2.500000  
3.000000, 3.500000, 4.000000, 4.500000, 5.000000, 5.500000  
6.000000, 6.500000, 7.000000, 7.500000, 8.000000, 8.500000  
9.000000, 9.500000, 10.000000, 10.500000, 11.000000, 11.500000

```
In[2]: #fmt="%d"为指定保存为整数  
np.savetxt("../tmp/arr.txt", arr, fmt="%d", delimiter=",")
```

```
In[3]: np.savetxt("../tmp/arr.txt", arr, fmt="%f", delimiter=",")
```



arr.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

0, 0, 1, 1, 2, 2  
3, 3, 4, 4, 5, 5  
6, 6, 7, 7, 8, 8  
9, 9, 10, 10, 11, 11

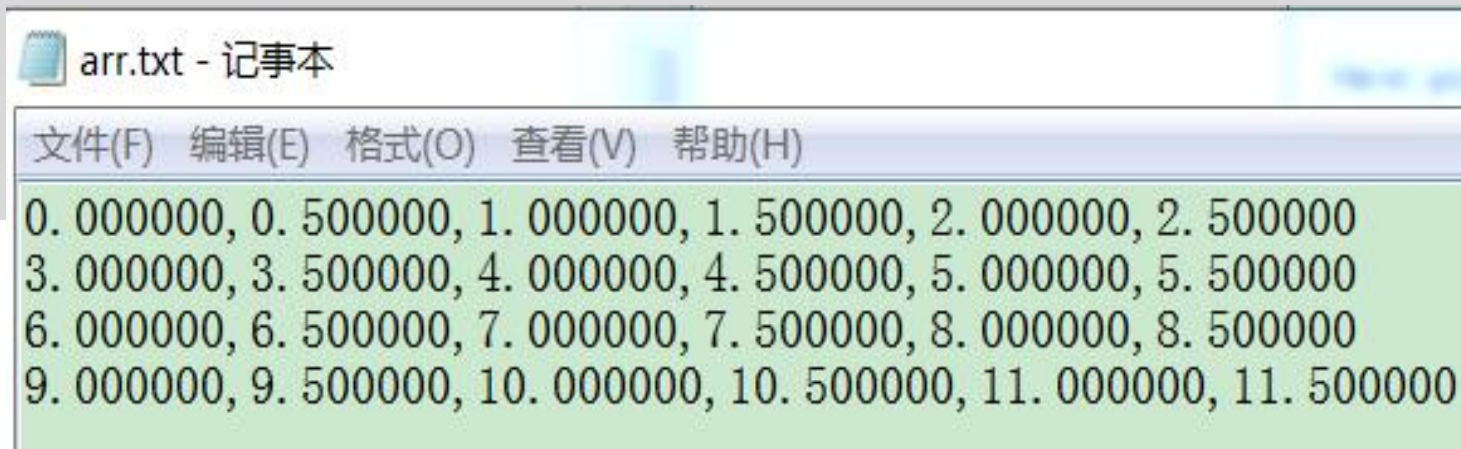


# 读写文件

## ➤ loadtxt函数执行的是把文件加载到一个二维数组中

```
In[1]: #读入的时候也需要指定逗号分隔
loaded_data = np.loadtxt("arr.txt", delimiter=",")
print('读取的数组为: \n', loaded_data)
```

```
Out[1]: 读取的数组为:
[[ 0.  0.5  1.  1.5  2.  2.5]
 [ 3.  3.5  4.  4.5  5.  5.5]
 [ 6.  6.5  7.  7.5  8.  8.5]
 [ 9.  9.5 10. 10.5 11. 11.5]]
```



arr.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
0.000000, 0.500000, 1.000000, 1.500000, 2.000000, 2.500000
3.000000, 3.500000, 4.000000, 4.500000, 5.000000, 5.500000
6.000000, 6.500000, 7.000000, 7.500000, 8.000000, 8.500000
9.000000, 9.500000, 10.000000, 10.500000, 11.000000, 11.500000
```

# 读写文件

➤ **genfromtxt**函数面向的是结构化数组和缺失数据。

```
In[1]: loaded_data = np.genfromtxt("arr.txt", delimiter = ",")  
print('读取的数组为: \n',loaded_data)
```

```
Out[1]: 读取的数组为:  
[[ 0.  0.5  1.  1.5  2.  2.5]  
 [ 3.  3.5  4.  4.5  5.  5.5]  
 [ 6.  6.5  7.  7.5  8.  8.5]  
 [ 9.  9.5 10. 10.5 11. 11.5]]
```

arr.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
0.000000, 0.500000, 1.000000, 1.500000, 2.000000, 2.500000  
3.000000, 3.500000, 4.000000, 4.500000, 5.000000, 5.500000  
6.000000, 6.500000, 7.000000, 7.500000, 8.000000, 8.500000  
9.000000, 9.500000, 10.000000, 10.500000, 11.000000, 11.500000
```

```
In[2]: loaded_data2 = np.genfromtxt("arr.txt", delimiter = (4,3,2))  
print('读取的数组为:',loaded_data2)
```

```
Out[2]: 读取的数组为:  
[[1234. 567. 89.]  
 [ 4.  7.  9.]  
 [ 45. 67.  9.]]
```

arr.txt - 记事本

文件(F) 编辑(E) 格式(O)

```
123456789  
 4  7  9  
4567 9
```

**genfromtxt**分割一个固定宽度的文件,列的宽度被定义为一个给定的字符数

# 使用函数进行简单统计分析

Numpy的排序方式主要可以概括为直接排序和间接排序两种

**直接排序：**只对数值直接进行排序

➤ **sort函数是最常用的排序方法：** *arr.sort()*

□ sort函数也可以指定一个axis参数，使得sort函数可以沿着指定轴对数据集进行排序

➤ axis=1为沿横轴排序； axis=0为沿纵轴排序。

**间接排序：**是指根据一个或多个键（key）对数据集进行排序

➤ **argsort函数返回值为重新排序值的下标：** *arr.argsort()*

➤ **lexsort函数返回值是按照最后一个传入数据排序的：** *np.lexsort((a,b,c))*

# 使用函数进行简单统计分析

## ➤ 使用sort函数进行排序

```
In[1]: np.random.seed(42)           #设置随机种子  
arr = np.random.randint(1,10,size = 10) #生成随机数  
print('创建的数组为: ',arr)
```

```
Out[1]: 创建的数组为: [7 4 8 5 7 3 7 8 5 4]
```

```
In[2]: arr.sort() #直接排序  
print('排序后数组为: ', arr)
```

```
Out[2]: 排序后数组为: [3 4 4 5 5 7 7 7 8 8]
```

# 使用函数进行简单统计分析

## ➤ 使用sort函数进行排序

```
In[3]: arr = np.random.randint(1,10,size = (3,3)) #生成3行3列的随机数  
print('创建的数组为: \n',arr)
```

```
Out[3]: 创建的数组为:  
[[8 8 3]  
 [6 5 2]  
 [8 6 2]]
```

```
In[4]: arr.sort(axis = 1) #沿着横轴排序  
print('排序后数组为: \n', arr)
```

```
Out[4]: 排序后数组为:  
[[3 8 8]  
 [2 5 6]  
 [2 6 8]]
```

```
In[5]: arr.sort(axis = 0) #沿着纵轴排序  
print('排序后数组为: ',arr)
```

```
Out[5]: 排序后数组为:  
[[2 5 6]  
 [2 6 8]  
 [3 8 8]]
```

# 使用函数进行简单统计分析

## ➤ 使用argsort函数进行排序

```
In[1]: arr = np.array([2,3,6,8,0,7])  
print('创建的数组为: ',arr)
```

```
Out[1]: 创建的数组为: [2 3 6 8 0 7]
```

```
In[2]: print('排序后数组为: ', arr.argsort()) #返回值为重新排序值的下标
```

```
Out[2]: 排序后数组为: [4 0 1 2 5 3]
```

# 使用函数进行简单统计分析

## ➤ 使用lexsort函数进行排序

|         |                                                                                                                                                                                       |                                        |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| In[1]:  | <pre>a = np.array([3,2,6,4,5]) b = np.array([50,30,40,20,10]) c = np.array([400,300,600,100,200])  #lexsort函数只接受一个参数，即(a,b,c)这样的元组，多个键值排序是按照最后一个传入数据计算的 d = np.lexsort((a,b,c))</pre> | 看起来像excel表格以某列为序（元组参数的最后一个元素），扩展排序整个表格 |
| Out[1]: | array([3, 4, 1, 0, 2], dtype=int64) #lexsort的返回值是最后一个数组c重新排序后的下标                                                                                                                      |                                        |
| In[2]:  | <pre>print('排序后数组为： ', list(zip(a[d],b[d],c[d])))</pre>                                                                                                                               |                                        |
| Out[2]: | 排序后数组为： [(4, 20, 100), (5, 10, 200), (2, 30, 300), (3, 50, 400), (6, 40, 600)]                                                                                                        |                                        |

# 使用函数进行简单统计分析

---

## 去重与重复数据

- 在统计分析的工作中，难免会出现“脏”数据的情况
- 重复数据就是“脏”数据的情况之一
- 如果一个一个的手动删除，耗时费力，效率低，容易出错
- 在NumPy中，可以通过**unique**函数找出数组中的唯一值并返回已排序的结果



# 使用函数进行简单统计分析

## 去重与重复数据

- 通过unique函数可以找出数组中的唯一值并返回已排序的结果

|         |                                                                                          |          |
|---------|------------------------------------------------------------------------------------------|----------|
| In[1]:  | names = np.array(['小明', '小黄', '小花', '小明', '小花', '小兰', '小白'])<br>print('创建的数组为: ', names) |          |
| Out[1]: | 创建的数组为: ['小明' '小黄' '小花' '小明' '小花' '小兰' '小白']                                             |          |
| In[2]:  | print('去重后的数组为: ', np.unique(names))                                                     |          |
| Out[2]: | 去重后的数组为: ['小兰' '小明' '小白' '小花' '小黄']                                                      |          |
| In[3]:  | ints = np.array([1,2,3,4,4,5,6,6,7,8,8,9,10])                                            | #创建数值型数据 |
|         | print('去重后的数组为: ', np.unique(ints))                                                      |          |
| Out[3]: | 去重后的数组为: [ 1  2  3  4  5  6  7  8  9 10]                                                 |          |

# 使用函数进行简单统计分析

## 去重与重复数据

- 另一种情况，在统计分析中也经常遇到，即需要把一个数据重复若干次。
- 在NumPy中主要使用**tile**函数和**repeat**函数实现数据重复。
- 使用tile函数实现数据重复： *np.tile(A, reps)*
  - 参数 “A” 指定重复的数组，参数 “reps” 指定重复的次数
- 使用repeat函数实现数据重复： *numpy.repeat(a, repeats, axis=None)*

# 使用函数进行简单统计分析

## 去重与重复数据

➤ 使用tile函数实现数据重复: *np.tile(A, reps)*

□ 参数 “A” 指定重复的数组, 参数 “reps” 指定重复的次数

```
In[1]: arr = np.arange(5)
      print('创建的数组为: ', arr)
```

```
Out[1]: 创建的数组为: [0 1 2 3 4]
```

```
In[2]: print('重复后数组为: ', np.tile(arr, 3)) #对数组进行重复
```

```
Out[2]: 重复后数组为: [0 1 2 3 4 0 1 2 3 4 0 1 2 3 4]
```

# 使用函数进行简单统计分析

## 去重与重复数据

- 使用repeat函数实现数据重复: *numpy.repeat(a, repeats, axis=None)*

```
In[1]: np.random.seed(42) #设置随机种子  
arr = np.random.randint(0,10,size = (3,3))  
print('创建的数组为: \n',arr)
```

```
Out[1]: 创建的数组为:  
[[6 3 7]  
 [4 6 9]  
 [2 6 7]]
```

```
In[2]: #按行进行元素重复  
print('重复后数组为: \n', arr.repeat(2, axis = 0))
```

```
Out[2]: 重复后数组为:  
[[6 3 7]  
 [6 3 7]  
 [4 6 9]  
 [4 6 9]  
 [2 6 7]  
 [2 6 7]]
```

# 使用函数进行简单统计分析

## ➤ 去重与重复数据

## ➤ 使用repeat函数实现数据重复

创建的数组为:

```
[[6 3 7]  
 [4 6 9]  
 [2 6 7]]
```

```
In[3]: print('重复后数组为: \n',arr.repeat(2, axis = 1)) #按列进行元素重复
```

```
Out[3]: 重复后数组为:  
[[6 6 3 3 7 7]  
 [4 4 6 6 9 9]  
 [2 2 6 6 7 7]]
```

## ➤ 这两个函数的主要区别在于

- ❑ tile函数是对数组进行重复操作
- ❑ repeat函数是对数组中的每个元素进行重复操作。

# 常用的统计函数

在NumPy中，有许多可以用于统计分析的函数。常见的统计函数有：

| 函数      | 说明          |
|---------|-------------|
| sum     | 计算数组的和      |
| mean    | 计算数组均值      |
| std     | 计算数组标准差     |
| var     | 计算数组方差      |
| min     | 计算数组最小值     |
| max     | 计算数组最大值     |
| argmin  | 返回数组最小元素的索引 |
| argmax  | 返回数组最大元素的索引 |
| cumsum  | 计算所有元素的累计和  |
| cumprod | 计算所有元素的累计积  |

- 几乎所有的统计函数在针对二维数组计算的时候都需要**注意轴的概念**。
- 当axis=0时，表示沿着纵轴进行计算。  
当axis=1时，表示沿着横轴计算。
- 但是在默认时，函数并不按照任一轴计算，而是计算一个总值。

# NumPy中常用统计函数的使用

## ➤ 创建实验数组

```
In[1]: arr = np.arange(20).reshape(4,5)
print('创建的数组arr为: \n', arr)
```

```
Out[1]: 创建的数组arr为:
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

# NumPy中常用统计函数的使用

## ➤ sum函数

```
In[2]: print('数组的和为: ', np.sum(arr)) #计算数组的和
```

```
Out[2]: 数组的和为: 190
```

```
In[3]: print('数组横轴的和为: ', arr.sum(axis = 0)) #沿着横轴计算求和
```

```
Out[3]: 数组横轴的和为: [30 34 38 42 46]
```

```
In[4]: print('数组纵轴的和为: ', arr.sum(axis = 1)) #沿着纵轴计算求和
```

```
Out[4]: 数组纵轴的和为: [10 35 60 85]
```

创建的数组arr为:

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



# NumPy中常用统计函数的使用

## ➤ mean函数

```
In[1]: print('数组的均值为: ', np.mean(arr))          #计算数组均值
```

```
Out[1]: 数组的均值为:  9.5
```

```
In[2]: print('数组横轴的均值为: ', arr.mean(axis = 0)) #沿着横轴计算数组均值
```

```
Out[2]: 数组横轴的均值为:  [ 7.5  8.5  9.5 10.5 11.5]
```

```
In[3]: print('数组纵轴的均值为: ', arr.mean(axis = 1)) #沿着纵轴计算数组均值
```

```
Out[3]: 数组纵轴的均值为:  [ 2.  7. 12. 17.]
```

创建的数组arr为:

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

# NumPy中常用统计函数的使用

## ➤ 标准差/方差函数

```
In[4]: print('数组的标准差为: ',np.std(arr))  #计算数组标准差
```

```
Out[4]: 数组的标准差为:  5.766281297335398
```

```
In[5]: print('数组的方差为: ',np.var(arr))  #计算数组方差
```

```
Out[5]: 数组的方差为:  33.25
```

**创建的数组arr为:**

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

# NumPy中常用统计函数的使用

## ➤ min/max/argmin/argmax函数

```
In[1]: print('数组的最小值为: ',np.min(arr)) #计算数组最小值
```

```
Out[1]: 数组的最小值为: 0
```

```
In[2]: print('数组的最大值为: ',np.max(arr)) #计算数组最大值
```

```
Out[2]: 数组的最大值为: 19
```

```
In[3]: print('数组的最小元素为: ',np.argmin(arr)) #返回数组最小元素的索引
```

```
Out[3]: 数组的最小元素为: 0
```

```
In[4]: print('数组的最大元素为: ',np.argmax(arr)) #返回数组最大元素的索引
```

```
Out[4]: 数组的最大元素为: 19
```

创建的数组arr为:

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

# NumPy中常用统计函数的使用

---

- **cumsum函数和cumprod函数**
- 之前介绍的函数计算都是聚合计算，直接产生计算的最终结果。
  - ▣ sum、mean、std、var、min、max等
- 在NumPy中有两个函数不是聚合计算：
  - ▣ cumsum函数和cumprod函数
  - ▣ 它们会产生一个由中间结果组成的数组

# NumPy中常用统计函数的使用

## ➤ cumsum函数和cumprod函数

```
In[1]: arr = np.arange(2,10)
       print('创建的数组为: ',arr)
```

```
Out[1]: 创建的数组为: [2 3 4 5 6 7 8 9]
```

```
In[2]: print('数组元素的累计和为: ', np.cumsum(arr)) #计算所有元素的累计和
```

```
Out[2]: 数组元素的累计和为: [ 2  5  9 14 20 27 35 44]
```

```
In[3]: print('数组元素的累计积为: ', np.cumprod(arr)) #计算所有元素的累计积
```

```
Out[3]: 数组元素的累计积为: [  2   6  24 120 720 5040 40320 362880]
```

# 任务实现

## 任务要求

- 读取iris数据集中的花萼长度数据（已保存为csv格式）
- 对其进行排序、去重
- 求出和、累积和、均值、标准差、方差、最小值、最大值



# 任务实现

## 1. 读取数据

```
In[1]: iris_sepal_length = np.loadtxt("iris_sepal_length.csv", delimiter=",") #读取文件
print('花萼长度表为：\n',iris_sepal_length)
```

```
Out[1]: 花萼长度表为：
[5.1 4.9 4.7 4.6 5.  5.4 4.6 5.  4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.  5.  5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.
 5.5 4.9 4.4 5.1 5.  4.5 4.4 5.  5.1 4.8 5.1 4.6 5.3 5.  7.  6.4 6.9 5.5
 6.5 5.7 6.3 4.9 6.6 5.2 5.  5.9 6.  6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
 6.3 6.1 6.4 6.6 6.8 6.7 6.  5.7 5.5 5.5 5.8 6.  5.4 6.  6.7 6.3 5.6 5.5
 5.5 6.1 5.8 5.  5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.  6.9 5.6 7.7 6.3 6.7 7.2
 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.  6.9 6.7 6.9 5.8 6.8
 6.7 6.7 6.3 6.5 6.2 5.9]
```

# 任务实现

## 2. 排序

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| In[2]:  | <pre>iris_sepal_length.sort()  #对数据进行排序 print('排序后的花萼长度表为: ',iris_sepal_length)</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Out[2]: | <pre>排序后的花萼长度表为: [4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7 4.7 4.8 4.8 4.8 4.8 4.8 4.9 4.9 4.9 4.9 4.9 4.9 5. 5. 5. 5. 5. 5. 5. 5. 5. 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.2 5.2 5.2 5.2 5.3 5.4 5.4 5.4 5.4 5.4 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.6 5.6 5.6 5.6 5.6 5.6 5.7 5.7 5.7 5.7 5.7 5.7 5.7 5.7 5.8 5.8 5.8 5.8 5.8 5.8 5.8 5.9 5.9 5.9 6. 6. 6. 6. 6. 6. 6.1 6.1 6.1 6.1 6.1 6.1 6.2 6.2 6.2 6.2 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.3 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.5 6.5 6.5 6.5 6.5 6.6 6.6 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.8 6.8 6.8 6.9 6.9 6.9 6.9 7. 7.1 7.2 7.2 7.2 7.3 7.4 7.6 7.7 7.7 7.7 7.7 7.9]</pre> |



# 任务实现

## 3. 去重、求和、求均值

|         |                                                                                                                                                        |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| In[3]:  | <code>print('去重后的花萼长度表为: ', np.unique(iris_sepal_length))</code>                                                                                       |
| Out[3]: | 去重后的花萼长度表为: [4.3 4.4 4.5 4.6 4.7 4.8 4.9 5. 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6. 6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7. 7.1 7.2 7.3 7.4 7.6 7.7 7.9] |
| In[4]:  | <code>print('花萼长度表的总和为: ', np.sum(iris_sepal_length)) #计算数组总和</code>                                                                                   |
| Out[4]: | 花萼长度表的总和为: 876.5                                                                                                                                       |
| In[5]:  | <code>print('花萼长度表的均值为: ', np.mean(iris_sepal_length)) #计算数组均值</code>                                                                                  |
| Out[5]: | 花萼长度表的均值为: 5.8433333333333334                                                                                                                          |

# 任务实现

## 4. 计算所有元素的累计和

In[6]:

```
print('花萼长度表的累计和为: \n', np.cumsum(iris_sepal_length))
```

Out[6]:

```
花萼长度表的累计和为:  
[ 4.3  8.7 13.1 17.5 22.  26.6 31.2 35.8 40.4 45.1 49.8 54.6  
 59.4 64.2 69.  73.8 78.7 83.6 88.5 93.4 98.3 103.2 108.2 113.2  
118.2 123.2 128.2 133.2 138.2 143.2 148.2 153.2 158.3 163.4 168.5 173.6  
178.7 183.8 188.9 194.  199.1 204.3 209.5 214.7 219.9 225.2 230.6 236.  
241.4 246.8 252.2 257.6 263.1 268.6 274.1 279.6 285.1 290.6 296.1 301.7  
307.3 312.9 318.5 324.1 329.7 335.4 341.1 346.8 352.5 358.2 363.9 369.6  
375.3 381.1 386.9 392.7 398.5 404.3 410.1 415.9 421.8 427.7 433.6 439.6  
445.6 451.6 457.6 463.6 469.6 475.7 481.8 487.9 494.  500.1 506.2 512.4  
518.6 524.8 531.  537.3 543.6 549.9 556.2 562.5 568.8 575.1 581.4 587.7  
594.1 600.5 606.9 613.3 619.7 626.1 632.5 639.  645.5 652.  658.5 665.  
671.6 678.2 684.9 691.6 698.3 705.  711.7 718.4 725.1 731.8 738.6 745.4  
752.2 759.1 766.  772.9 779.8 786.8 793.9 801.1 808.3 815.5 822.8 830.2  
837.8 845.5 853.2 860.9 868.6 876.5]
```

# 任务实现

## 5. 求标准差、方差、最大值、最小值

```
In[7]: print('花萼长度表的标准差为: ', np.std(iris_sepal_length))
```

```
Out[7]: 花萼长度表的标准差为: 0.8253012917851409
```

```
In[8]: print('花萼长度表的方差为: ', np.var(iris_sepal_length)) #计算数组方差
```

```
Out[8]: 花萼长度表的方差为: 0.6811222222222223
```

```
In[9]: print('花萼长度表的最大值为: ', np.max(iris_sepal_length)) #计算最大值
```

```
Out[9]: 花萼长度表的最大值为: 7.9
```

```
In[10]: print('花萼长度表的最小值为: ', np.min(iris_sepal_length)) #计算最小值
```

```
Out[10]: 花萼长度表的最小值为: 4.3
```



大数据，成就未来

# Thank you!

