

Notepad of <Python for Data Analysis>

Numpy

Kitsu Liu

Graduate School of Environment and Information Sciences

Yokohama National University

目录

1	基础	3
2	索引与切片	10
2.1	扩展切片	15
2.1.1	<code>step</code> 为正数	15
2.1.2	<code>step</code> 为负数	16
3	数组的数据结构以及变更	17
4	数组操作	20
4.1	数值操作	20
4.2	数组排序	22
4.3	排序的索引	23
5	数组形状操作	24
5.1	改变形状	24
5.2	单纯的增加维度（增加空维度）	26
5.3	减去空维度	28
5.4	矩阵的转置	29
5.5	数组的连接	30
5.6	把矩阵拉平成向量	32
6	数组生成	34

6.0.1 灵活应用：	39
7 运算	42
7.1 一般运算	42
7.2 统计量的运算	46
7.2.1 最大值	47
7.2.2 最小值	48
7.2.3 找最值的索引	49
7.2.4 平均值	49
7.2.5 標準偏差	50
7.2.6 分散	51
8 随机模块	52
8.1 生成随机浮点数	53
8.2 生成随机整数	53
8.3 生成服从正态分布 (高斯分布, ガウス分布, 正規分布) 的随机数	54
8.4 洗牌 (打乱顺序)	54
8.5 乱数の初期化 (seed)	55
9 文件读写	57
9.1 TXT 类型文件：	59
9.1.1 从外面读取进 numpy,python	59
9.1.2 从 Numpy 里导出：	61
9.2 npy 和 npz 类型文件：	62
9.2.1 从 Numpy 里导出：	62
9.2.2 从外面读取进 numpy,python	62
10 Numpy 练习题	64

1 基础

```
[1]: import numpy as np
# 查看有没有安装 numpy 后加载并在引用时简称为 np
```

```
[6]: array = [1,2,3,4,5]
array +1
# 一般的 array 不可以直接加一
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-6-f308850a6c21> in <module>

TypeError: can only concatenate list (not "int") to list
```

```
[14]: array = np.array([1,2,3,4,5])
# 用 numpy 去生成 array 的结构
print(type(array))
# 看一下生成的 array 的结构分类
array
# 打印
```

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

```
[14]: array([1, 2, 3, 4, 5])
```

```
[15]: array += 1
# 因为是 numpy 生成的 ndarray 结构，所以可以直接 +1
array
# 打印
```

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

```
[16]: array2 = array +1
# 生成一个新的 ndarray，为 array+1
array2
```

```
[16]: array([3, 4, 5, 6, 7])
```

```
[17]: array2 + array
      # 把数列相加，结果为对应位置元素的相加
```

```
[17]: array([ 5,  7,  9, 11, 13])
```

```
[18]: array2 * array
      # 把数列相乘，结果为对应位置元素的相乘，有点类似于向量乘法，但是并没有做求和
```

```
[18]: array([ 6, 12, 20, 30, 42])
```

```
[19]: array[0]
      # 从 array 取值，数列是从 0 开始，不是从 1 开始
```

```
[19]: 2
```

```
[20]: array.shape
      # 查看矩阵的各维度元素数量，下面的是典型的一维数列，所以后面的逗号都是空的
```

```
[20]: (5,)
```

```
[22]: tang_list = [1,2,3,4,5]
      # 生成一个一般的 array
      tang_list.shape
      #xxx.shape 不可用于一般 array
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-22-c50275e90157> in <module>

AttributeError: 'list' object has no attribute 'shape'
```

```
[56]: array3 = np.array([[1,2,3],[4,5,6]])
      # 用 numpy 把一个 list of list 结构转换为 ndarray 的结构，然后就可以表示矩阵了
      # 这里是一个 2*3 的矩阵，两行三列

      print(type(array3))
      # 看一下生成的 array3 的结构分类
```

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

```
[31]: array3
# 这里是一个 2*3 的矩阵，两行三列
```

```
[31]: array([[1, 2, 3],
            [4, 5, 6]])
```

```
[33]: array3.shape
# 看一下各维度的元素数量，也就是看一下是几行几列的矩阵。
# 输出表示为（行数，列数）
```

```
[33]: (2, 3)
```

```
[63]: tang_list = [1,2,3,4,5]
# 生成一个一般的 list

tang_array = np.array(tang_list)
# 用 np.array 进行转换

tang_array
# 打印
```

```
[63]: array([1, 2, 3, 4, 5])
```

```
[36]: type(tang_array)
# 看一下它的结构类型
```

```
[36]: numpy.ndarray
```

```
[37]: tang_array.dtype
# xxx.dtype 是看一下 array 里面的元素类型
```

```
[37]: dtype('int64')
```

```
[38]: tang_list2 = [1,2,3,4,5.0]
      tang_array2 = np.array(tang_list2)
      # 注意：把最后一个改成了 “5.0”
      tang_array2.dtype
```

```
[38]: dtype('float64')
```

注意：这个输出结果变成了 float64，不是 int64 了

也就是说 ndarray 是要求所有的元素都是同一类型
但是前面的 1, 2, 3, 4 都是 **int**，只有最后的 5.0 是 **float**
所以要求兼容，而 ‘小数 float’ 的适用范围 > ‘整数 int’
所以就被向下对齐，向下兼容

```
[40]: tang_list3 = [1,2,3,4,'5']
      tang_array3 = np.array(tang_list3)
      # 注意：把最后一个改成了 '5'

      tang_array3.dtype
```

```
[40]: dtype('<U21')
```

同理，这里的 **String** 类型的 '5'
也把别的元素变成的 String 类型

```
[46]: tang_array.itemsize
      #xxx.itemsize 用于查看数列中每一个元素占了多大的内存。单位是 Byte
```

```
[46]: 8
```

itemsize 的计算原理

tang_array.itemsize 输出的是 8

而 tang_array.dtype 输出的是 int64

也就是说，tang_array 是里面每一个元素是一个 64 位的整数，也就是 $64=2^6$ 位 (bit)

而 $1\text{Byte} = 8\text{bit} = 2^3\text{bit}(\text{位})$

所以 $2^6 / 2^3 = 2^3 = 8$

```
[57]: print("矩阵 array3 为: ")
      print(array3)

      print("size 的结果 =",array3.size)
      print("shape 的结果 =",array3.shape)
```

矩阵 array3 为:

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

size 的结果 = 6

shape 的结果 = (2, 3)

xxx.size 打印有所有维度中元素的数量，并不去区分有几个维度

xxx.shape 会区分维度

```
[75]: print("矩阵 array3 为: ")
      print(array3)
      print("向量 tang_array 为: ")
      print(tang_array)
      print("矩阵 array3 的维度 =",array3.ndim)
      print("tang_array 的维度 =",tang_array.ndim)

      #xxx.ndim 用于看维度
```

矩阵 array3 为:

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

向量 tang_array 为:

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

矩阵 array3 的维度 = 2

tang_array 的维度 = 1

xxx.ndim 用于看维度

[86]: `xxxx.fill`(任意数字) 用 ‘任意数字’ 去覆盖性的填充矩阵/向量

```
print("原矩阵为: ")
print(array4)

print("用.fill(0) 去填充后: ")
array5 = np.array([[1,2,3],[4,5,6],[7,8,9]])
array5.fill(0)
print(array5)

print("用.fill(1) 去填充后: ")
array6 = np.array([[1,2,3],[4,5,6],[7,8,9]])
array6.fill(1)
print(array6)
```

原矩阵为:

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

用.fill(0) 去填充后:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
```

用.fill(1) 去填充后:

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

`xxxx.fill`(任意数字) 用 ‘任意数字’ 去覆盖性的填充矩阵/向量
但是要注意是覆盖性填充, 会破坏原有的内容


```
[84]: array4 = np.array([[1,2,3],[4,5,6],[7,8,9]])  
      # 新整一个矩阵，方便做实验  
      print("新矩阵 array4 为: ")  
      print(array4)  
  
      print("矩阵 array4 的维度 =",array4.ndim)  
      print("nsize 的结果 =",array4.size)  
      print("shape 的结果 =",array4.shape)
```

新矩阵 array4 为:

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

矩阵 array4 的维度 = 2

nsize 的结果 = 9

shape 的结果 = (3, 3)

2 索引与切片

```
[99]: print(tang_array)
      print(tang_array[0])
      print(tang_array[3])
      print(tang_array[0:3])
      print(tang_array[1:3])
      print(tang_array[-2:])
      print(tang_array[-4:])
      print(tang_array[-4:4])
      print(tang_array[:-1])
```

数列/向量为:

[1 2 3 4 5]

索引, 取第 0 个值:

1

索引, 取第 3 个值:

4

切片, 取第 0 个值到第 2 个值:

[1 2 3]

切片, 取第 1 个值到第 2 个值:

[2 3]

切片, 从倒数第 2 个值取到最后一个:

[4 5]

切片, 从倒数第 4 个值取到最后一个:

[2 3 4 5]

切片, 从倒数第 4 个值取到第 3 个:

[2 3 4]

切片, 从第 0 个值 (也就是最开始的值) 取到倒数第二个:

[1 2 3 4]

```
[102]: print("矩阵为: ")
        print(array4)

        print("(2,3) 元素为: ",array4[1,2])
        # 取第 a+1 行, 第 b+1 列的元素, 即取 (a+1,b+1) 元素
        # 写为: array[a,b], 这里的 array 是矩阵名
```

矩阵为:

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

(2,3) 元素为: 6

但是要注意!!

矩阵的开始也是从第 0 行, 第 0 列开始!!!

```
[104]: print("原矩阵为: ")
        print(array4)

        print("给 (2,3) 元素的值改为 99: ")
        array7 = np.array([[1,2,3],[4,5,6],[7,8,9]])
        array7[1,2] = 99    # 给 (2,3) 元素的值改为 99
        print(array7)
```

原矩阵为:

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

给 (2,3) 元素的值改为 99:

```
[[ 1  2  3]
 [ 4  5 99]
 [ 7  8  9]]
```

```
[107]: print(array4)

print("单独取第一行: ",array4[0])
print("单独取第二行: ",array4[1])
print("单独取第三行: ",array4[2])
用 array[x] 可以从矩阵中来取任意第 x 行

print("单独取第一列: ",array4[:,0])
print("单独取第二列: ",array4[:,1])
print("单独取第三列: ",array4[:,2])
用 array[:,y] 可以从矩阵中来取任意第 y 列
# 底层逻辑: 【先从最开始的一行取到最后的一行, 也就是取所有的行】
# 在那个基础上只留下在各行的第 y 列的元素
```

原矩阵为:

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

单独取第一行: [1 2 3]

单独取第二行: [4 5 6]

单独取第三行: [7 8 9]

用 array[x] 可以从矩阵中来取任意第 x 行

单独取第一列: [1 4 7]

单独取第二列: [2 5 8]

单独取第三列: [3 6 9]

用 array[:,y] 可以从矩阵中来取任意第 y 列

但是要注意!!

矩阵的开始也是从第 0 行, 第 0 列开始!!!

```
[111]: array8 = np.array([[1,2,3],[4,5,6],[7,8,9]])
        print(array8)
        array9 = array8
        print(array9)
        array9[1,1] = 100 # 让矩阵 array9 的 (2,2) 元素变为 100
        print(array9)
        print(array8)
```

python 中走的是类似指针的感觉，不会再单独在内存里开一个新的

原矩阵为：

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

设一个新的 array9，用 array8 给它赋值

array9 为：

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

更改后的 array9 为：

```
[[ 1  2  3]
 [ 4 100 6]
 [ 7  8  9]]
```

注意！此时的 array8 为：

```
[[ 1  2  3]
 [ 4 100 6]
 [ 7  8  9]]
```

如果是按 C 里面的逻辑的话，array8 的值不会变的，但是在 python 中是类似于指针的感觉，共享内存中同一块位置，不会单独再开新的

```
[112]: array8 = np.array([[1,2,3],[4,5,6],[7,8,9]])
        print(array8)
        array9 = array8.copy()
        print(array9)
        array9[1,1] = 100
        # 让矩阵 array9 的 (2,2) 元素变为 100
        print(array9)
        print(array8)
```

但是用 `xxx.copy()`，这个参数命令，就会在内存里重新分配一个位置

原矩阵为：

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

设一个新的 `array9`，用 `array8` 给它赋值

`array9` 为：

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

更改后的 `array9` 为：

```
[[ 1  2  3]
 [ 4 100 6]
 [ 7  8  9]]
```

注意！此时的 `array8` 为：

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

```
[110]: print(array4)
       print(array4[0:2,1:3])
```

接上面的高级玩法，考虑取它的一部分，也就是取余子式的那种感觉，取子矩阵

原矩阵为：

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

取从第 0 行到第 1 行，第 1 列到第 2 列的子矩阵：

```
[[2 3]
 [5 6]]
```

2.1 扩展切片

`a[start:stop:step]`，其中 `step` 是一个非零整数，即比简单切片多了调整步长的功能，此时切片的行为可概括为：从 `start` 对应的位置出发，以 `step` 为步长索引序列，直至越过 `stop` 对应的位置，且不包括 `stop` 本身

事实上，简单切片就是 `step=1` 的扩展切片的特殊情况。需要详细解释的是 `step` 分别为正数和负数的两种情况

具体全文详见：[click this](#)

2.1.1 `step` 为正数

当 `step` 为正数时，切片行为很容易理解，`start` 和 `stop` 的截断和缺省规则也与简单切片完全一致：

```
[64]: tang_array
```

```
[64]: array([1, 2, 3, 4, 5])
```

```
[66]: tang_array[0:5:2]
```

```
[66]: array([1, 3, 5])
```

```
[67]: tang_array[::2]
```

```
[67]: array([1, 3, 5])
```

```
[68]: tang_array[:-2:2]
```

```
[68]: array([1, 3])
```

```
[69]: tang_array[1::2]
```

```
[69]: array([2, 4])
```

2.1.2 step 为负数

当 **step** 为负数时，切片将其解释为从 **start** 出发以步长 **|step|** **逆序** 索引序列，此时，**start** 和 **stop** 的截断依然遵循前述规则，但缺省发生一点变化，因为我们说过，在缺省的情况下，Python 的行为是尽可能取最大区间，此时访问是**逆序**的，**start** 应尽量取大，**stop** 应尽量取小，才能保证区间最大，因此：

按照扩充索引范围的观点，**start** 的缺省值是无穷大，**stop** 的缺省值是无穷小

```
[70]: tang_array
```

```
[70]: array([1, 2, 3, 4, 5])
```

```
[73]: tang_array[4::-1]  
# 从第 4 个开始往前，间隔 1 取到 0
```

```
[73]: array([5, 4, 3, 2, 1])
```

```
[74]: tang_array[::-1]  
# 将数组倒序
```

```
[74]: array([5, 4, 3, 2, 1])
```


3 数组的数据结构以及变更

```
[116]: mask = np.array([0,0,0,1,1,1,0,0,1,1],dtype = bool)# 生成一个由布尔值构成的数组
print(mask)
print("看一下我们要用的数组 array10",array10)
array10[mask]
print(array10[mask])
```

用布尔值构成的数组可以作为索引去取值

使用 `mask = np.array([0,0,0,1,1,1,0,0,1,1],dtype = bool)` 生成的数组为:

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

看一下我们要用的数组 `array10 [0 10 20 30 40 50 60 70 80 90]`

使用为 `array10[mask]`, 也就是将布尔值数组 `mask` 传进 `array10` 后看看会怎么样:

```
[30 40 50 80 90]
```

看一下我们会发现在 `true` 的位置对应出来的地方, 对照 `array10` 的值被取出来了
但是要注意, 布尔值的数组中元素数量必须与要取的数组元素数量相等, 否则会报错

```
[119]: random_array=np.random.rand(10)
print(random_array)# 生成 0-1 之间的 10 个数字
print(mask)
mask = random_array > 0.5
print(mask)
```

用 `np.random.rand(xx)`, 可以生成在【0 到 1 之间】的 `xx` 个数

```
[0.04939647 0.31370702 0.42854165 0.77241394 0.9834424  0.15768831
 0.82184587 0.54881796 0.51759371 0.59256235]
```

使用 `mask = np.array([0,0,0,1,1,1,0,0,1,1],dtype = bool)` 生成的数组为:

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

看一下我们使用 `random.array` 去批量更改 `mask` 的值:

```
mask = random_array > 0.5
```

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

```
[123]: array11 = np.array([10,20,30,40,50])
print("array11 =",array11)

print(array11 > 30)

print(np.where(array11 > 30))
```

使用 `np.where()` 查找索引（限定条件）

```
array11 = [10 20 30 40 50]
```

看一下各个元素的值是否大于 30, `array11 > 30`

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

使用 `np.where(array11 > 30)` 来验证一下
(`array([3, 4])`),)

数组从第 0 个开始的, 结果显示第 3 个和第 4 个值是大于 30 的

```
[126]: array12 = np.array([1,2,3,4,5],dtype = np.float64)
print("array12 =",array12)

print("array12.dtype =",array12.dtype)

print("array12.nbytes =",array12.nbytes,"Bytes")
```

使用 `array12 = np.array([1,2,3,4,5],dtype = np.float64)` 生成数组
`np.float64` 意为: Numpy 中的 64 位浮点小数

```
array12 = [1. 2. 3. 4. 5.]
```

使用 `array12.dtype` 来看一下它的类型
`array12.dtype = float64`

使用 `array12.nbytes` 来看一下它占了多少内存
`array12.nbytes = 40 Bytes`

```
[133]: array13 = np.array([1,2,3,4,'str'],dtype = np.object)
        print("array13 =",array13)

        print("array13 * 2 =",array13 * 2)
```

使用 `array13 = np.array([1,2,3,4,'str'],dtype = np.object)` 生成数组
`np.object` 意为：Numpy 中的 object

```
array13 = [1 2 3 4 'str']
```

`array13 * 2`，乘二试一下

```
array13 * 2 = [2 4 6 8 'strstr']
```

发现数字就直接乘 2 了，但是文字是直接多复制了一份

```
[167]: print("原 array11 为",array11)
        print("各个元素的数据类型为",array11.dtype)

        array11 = np.asarray(array11,dtype = np.float64)
        print("更改后各个元素的数据类型为",array11.dtype)

        array11 = array11.astype(np.int64)
        print("更改后各个元素的数据类型为",array11.dtype)
```

使用 `np.asarray(array_name,dtype = np.xxxx)`，可以更改数组的各元素的数据类型
 比如：

原 `array11` 为 `[10 20 30 40 50]`

各个元素的数据类型为 `int64`

使用 `array11 = np.asarray(array11,dtype = np.float64)` 改一下试试

更改后各个元素的数据类型为 `float64`

还可以使用 `np.asarray = np.asarray.astype(np.xxxx)`，可以更改数组的各元素的数据类型

使用 `array11 = array11.astype(np.int64)` 改一下试试

更改后各个元素的数据类型为 `int64`

但是要注意，单独使用 `astype` 和 `np.asarray` 都不会更改原来的数组的数据类型

所以想要更改的话，就要写 `array_name = np.asarray(array_name,dtype = np.xxxx)`

以及 `array_name = array_name.astype(np.xxxx)`

4 数组操作

4.1 数值操作

限定范围（让大于 y 的等于 y ，小于 x 的等于 x ）

但是并不会更改原数组（矩阵）

```
array__name.clip(x,y)
```

四舍五入

但是并不会更改原数组（矩阵）

```
array__name.round(参数) 以及 np.round(array__name, 参数)
```

参数可写：`decimals = xx`，意为四舍五入到小数点后第几位

如果不写参数则默认为四舍五入到整数位

以上都不会更改原数组（矩阵），若想更改原数组（矩阵）则需要写成如下所示

```
array__name = array__name.clip(x,y)
```

```
array__name = array__name.round(参数)
```

```
array__name = np.round(array__name, 参数)
```

```
[192]: array13
```

```
[192]: array([[1, 2, 3],  
            [4, 5, 6]])
```

```
[190]: array13.clip(2,4)  
# 让所有元素，小于 2 的使其等于 2，大于 4 的使其等于 4
```

```
[190]: array([[2, 2, 3],  
            [4, 4, 4]])
```

```
[193]: array14 = np.array([[1.5, 2.865, 3.145], [4.843, 5.1478, 6.92], [7.18, 8.0, 9.48]])  
array14
```

```
[193]: array([[1.5    , 2.865  , 3.145  ],  
            [4.843  , 5.1478, 6.92   ],  
            [7.18   , 8.     , 9.48   ]])
```

```
[196]: array14.dtype
```

```
[196]: dtype('float64')
```

```
[197]: array14.round()  
## 全局四舍五入，四舍五入到整数位
```

```
[197]: array([[2., 3., 3.],  
             [5., 5., 7.],  
             [7., 8., 9.]])
```

```
[198]: np.round(array14)  
## 全局四舍五入，四舍五入到整数位
```

```
[198]: array([[2., 3., 3.],  
             [5., 5., 7.],  
             [7., 8., 9.]])
```

```
[201]: array14.round(decimals = 1)  
## 全局四舍五入，四舍五入到小数点后第一位
```

```
[201]: array([[1.5, 2.9, 3.1],  
             [4.8, 5.1, 6.9],  
             [7.2, 8. , 9.5]])
```

```
[202]: np.round(array14, decimals = 3)  
## 全局四舍五入，四舍五入到小数点后第二位
```

```
[202]: array([[1.5   , 2.865, 3.145],  
             [4.843, 5.148, 6.92 ],  
             [7.18  , 8.   , 9.48 ]])
```

```
[204]: np.round(array14, decimals = 0)  
## 全局四舍五入，四舍五入到小数点后第 0 位，想到入四舍五入到整数位
```

```
[204]: array([[2., 3., 3.],  
             [5., 5., 7.],  
             [7., 8., 9.]])
```

4.2 数组排序

排序（默认升序，从小到大）

`np.sort(array_name, 参数)`

参数可写：`axis = xx` 如果不写参数默认排的是行向量内部（也就是横着的内部）

的效果不是 `axis = 0`，而是 `axis = -1`

写 `axis = 0` 则会排竖着的列向量的内部。

这一点跟之前的维度的理解不太一样，需要特别记忆

```
[209]: array15 = np.array([[1.5,1.3,7.5],[5.6,1.2,6.6]])  
array15
```

```
[209]: array([[1.5, 1.3, 7.5],  
             [5.6, 1.2, 6.6]])
```

```
[210]: np.sort(array15)  
# 默认的排序，升序，仅影响【行】向量内部，不影响向量之间
```

```
[210]: array([[1.3, 1.5, 7.5],  
             [1.2, 5.6, 6.6]])
```

```
[211]: np.sort(array15,axis = 0)  
# 升序，仅影响【列】向量内部，不影响向量之间
```

```
[211]: array([[1.5, 1.2, 6.6],  
             [5.6, 1.3, 7.5]])
```

```
[212]: np.sort(array15,axis = 1)  
# 效果等同于默认
```

```
[212]: array([[1.3, 1.5, 7.5],  
             [1.2, 5.6, 6.6]])
```

```
[213]: np.sort(array15,axis = -1)  
# 效果等同于默认
```

```
[213]: array([[1.3, 1.5, 7.5],  
             [1.2, 5.6, 6.6]])
```

4.3 排序的索引

可以用 `np.argsort(array_name, 参数)` 来看数组中各个向量中的大小索引（大小关系）

参数可写 `axis = xx`

如果不写参数默认排的是行向量内部（也就是横着的内部）

的效果不是 `axis = 0`，而是 `axis = -1`

写 `axis = 0` 则会排竖着的列向量的内部

这一点跟之前的维度的理解不太一样，需要特别记忆

```
[214]: array15
```

```
[214]: array([[1.5, 1.3, 7.5],  
            [5.6, 1.2, 6.6]])
```

```
[216]: np.argsort(array15)  
# 看一下各个【行】向量内元素的大小情况，对应位置的谁排第几  
# 注意!!!! 0是最小
```

```
[216]: array([[1, 0, 2],  
            [1, 0, 2]])
```

```
[217]: np.argsort(array15,axis = 0)  
# 看一下各个【列】向量内元素的大小情况，对应位置的谁排第几  
# 注意!!!! 0是最小
```

```
[217]: array([[0, 1, 1],  
            [1, 0, 0]])
```

```
[218]: np.argsort(array15,axis = -1)  
# 与默认相同
```

```
[218]: array([[1, 0, 2],  
            [1, 0, 2]])
```

```
[219]: np.argsort(array15,axis = -1)  
# 与默认相同
```

```
[219]: array([[1, 0, 2],  
            [1, 0, 2]])
```

5 数组形状操作

5.1 改变形状

用 `array_name.shape = x,y` 以及 `array_name.reshape(x,y)`

可以把 `array_name` 这个数组（矩阵/向量）改变成【 x 乘 y 】形状的（矩阵/向量），也就是 x 行 y 列

但是有一个限制条件，就是 x 乘 y 必须等于数组中的元素的数量

比如下面的例子中，`array16` 有 10 个元素，那么就算变形也只能变成【2 5】，【5 2】，【10 1】除此以外就不行了。

此外，`array_name.shape = x,y` 会更改原数组，

但是 `array_name.reshape(x,y)` 不会

```
[21]: array16 = np.arange(0,20,2)
      #0-20 每间隔 2 生成一个数，以这样的规则生成一个数列，但是不包括右边的限值 20
      array16
```

```
[21]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
[250]: array16.shape
```

```
[250]: (10,)
```

```
[252]: array16.shape = 5,2
      array16
```

```
[252]: array([[ 0,  2],
              [ 4,  6],
              [ 8, 10],
              [12, 14],
              [16, 18]])
```

```
[253]: array16.shape = 2,5
      array16
```

```
[253]: array([[ 0,  2,  4,  6,  8],
              [10, 12, 14, 16, 18]])
```



```
[254]: array16.shape = 10,1  
array16
```

```
[254]: array([[ 0],  
             [ 2],  
             [ 4],  
             [ 6],  
             [ 8],  
            [10],  
            [12],  
            [14],  
            [16],  
            [18]])
```

```
[256]: array16.reshape(5,2)  
#array_name.reshape(不会更改原数组)
```

```
[256]: array([[ 0,  2],  
             [ 4,  6],  
             [ 8, 10],  
            [12, 14],  
            [16, 18]])
```

```
[259]: array16  
# 依旧是上面 IN[254] 的结果
```

```
[259]: array([[ 0],  
             [ 2],  
             [ 4],  
             [ 6],  
             [ 8],  
            [10],  
            [12],  
            [14],  
            [16],  
            [18]])
```

```
[261]: array16 = array16.reshape(2,5)
# 但是这样就会改变原数组了
array16
```

```
[261]: array([[ 0,  2,  4,  6,  8],
              [10, 12, 14, 16, 18]])
```

5.2 单纯的增加维度（增加空维度）

使用 `array_name[x:y,np.newaxis]` 以及 `array_name[np.newaxis,x:y]` 来增加维度

其中 `x:y` 是切片，表示要原来数组中对应位置维度的 `x` 到 `y-1` 项

如果直接写成 `[:,np.newaxis]` 或者是 `[np.newaxis,:]` 的话就是对应位置的维度里的所有元素都要
当然，可以加任意个 `np.newaxis`，加几个就是加几维

另外，此命令【不会更改】原数组。

```
[20]: array16.shape = 10,
array16
# 先看一下原数组
```

```
[20]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
[267]: array16.shape
```

```
[267]: (10,)
```

```
[288]: print(array16[np.newaxis,:])
print(" ")
print(array16[np.newaxis,:].shape)
# 虽然这样增加了维度
# 从前面
```

```
[[ 0  2  4  6  8 10 12 14 16 18]]
```

```
(1, 10)
```

```
[289]: print(array16)
       print(array16.shape)
       # 但是原数组没有被更改
```

```
[ 0  2  4  6  8 10 12 14 16 18]
```

```
(10,)
```

```
[290]: array16 = array16[np.newaxis,:]  
       # 这样写就会改变原数组了  
       print(array16)  
       print(array16.shape)
```

```
[[ 0  2  4  6  8 10 12 14 16 18]]
```

```
(1, 10)
```

```
[291]: array16 = array16.reshape(10,)  
       array16  
       # 先还原一下
```

```
[291]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
[292]: print(array16[:,np.newaxis])  
       # 从后面增加维度  
       print(array16[:,np.newaxis].shape)
```

```
[[ 0]
```

```
 [ 2]
```

```
 [ 4]
```

```
 [ 6]
```

```
 [ 8]
```

```
[10]
```

```
[12]
```

```
[14]
```

```
[16]
```

```
[18]]
```

```
(10, 1)
```

5.3 减去空维度

使用 `array_namep.squeeze()` 来减去【所有】空维度

另外，此命令【不会更改】原数组。

```
[6]: array16  
# 打印一下原数组
```

```
[6]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
[7]: array16.shape
```

```
[7]: (10,)
```

```
[12]: array16 = array16[np.newaxis,:]  
# 增加一个空维度  
array16.shape
```

```
[12]: (1, 10)
```

```
[13]: array16 = array16.squeeze()  
# 减去所有空维度  
array16.shape
```

```
[13]: (10,)
```

```
[14]: array16 = array16[np.newaxis,np.newaxis,np.newaxis,:,np.newaxis,np.newaxis,]  
# 增加五个空维度  
array16.shape
```

```
[14]: (1, 1, 1, 10, 1, 1)
```

```
[15]: array16 = array16.squeeze()  
# 减去所有空维度  
array16.shape
```

```
[15]: (10,)
```

5.4 矩阵的转置

使用 `array_name.transpose()` 来转置矩阵

还可以使用简略版的 `array_name.T` 来转置矩阵

另外，此命令【不会更改】原数组。

```
[22]: array16.shape = 2,5 #先做成 2*5 的矩阵，这样下面会比较清晰
array16
```

```
[22]: array([[ 0,  2,  4,  6,  8],
            [10, 12, 14, 16, 18]])
```

```
[26]: array16.transpose() #转置了
```

```
[26]: array([[ 0, 10],
            [ 2, 12],
            [ 4, 14],
            [ 6, 16],
            [ 8, 18]])
```

```
[28]: array16 #但是原矩阵没有被改变
```

```
[28]: array([[ 0,  2,  4,  6,  8],
            [10, 12, 14, 16, 18]])
```

```
[29]: array16.T #转置了
```

```
[29]: array([[ 0, 10],
            [ 2, 12],
            [ 4, 14],
            [ 6, 16],
            [ 8, 18]])
```

```
[30]: array16 #但是原矩阵没有被改变
```

```
[30]: array([[ 0,  2,  4,  6,  8],
            [10, 12, 14, 16, 18]])
```

5.5 数组的连接

使用 `np.concatenate((array__name1,array__name2), 参数)` 把矩阵拼起来

但是要注意，上面是 `(())` 双层括号，第一个括号表示传参数，第二个表示生成出来的是个数组

参数可写：`axis = xx` 意为在维度 `xx` 上把数组拼起来，注：不写参数的话，就是默认是 `axis = 0`

另外，此命令【不会更改】原数组。

```
[32]: array8 = np.array([[1,2,3],[4,5,6],[7,8,9]])
      array9 = np.array([[11,21,31],[41,51,61],[71,81,91]])
      array10 = np.array([[1,2],[4,5],[7,8]])
```

```
[34]: print(array8)
      print(array9)
```

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

```
[[11 21 31]
 [41 51 61]
 [71 81 91]]
```

array8 和 array9 都是 3*3 的矩阵

```
[35]: array10 #array10 是 3*2 的矩阵
```

```
[35]: array([[1, 2],
            [4, 5],
            [7, 8]])
```

```
[46]: np.concatenate((array8,array9))
      # 把 array8 和 array9 拼一起。8 在前，9 在后，因为上面的代码就是 8 在前
```

```
[46]: array([[ 1,  2,  3],
            [ 4,  5,  6],
            [ 7,  8,  9],
            [11, 21, 31],
            [41, 51, 61],
            [71, 81, 91]])
```

```
[48]: np.concatenate((array8,array9),axis = 0)
# 与默认一致，就是在第 0 维拼，也就是简单的【竖着拼】，把【行向量】竖着拼在一起
```

```
[48]: array([[ 1,  2,  3],
            [ 4,  5,  6],
            [ 7,  8,  9],
            [11, 21, 31],
            [41, 51, 61],
            [71, 81, 91]])
```

```
[49]: np.concatenate((array8,array9),axis = 1)
## 在第 1 维拼，【横着拼】，把【列向量】横着拼在一起
```

```
[49]: array([[ 1,  2,  3, 11, 21, 31],
            [ 4,  5,  6, 41, 51, 61],
            [ 7,  8,  9, 71, 81, 91]])
```

```
[43]: np.concatenate((array8,array10))
## 会发现 array8 和 array10 没法在第 0 维拼在一起
# 第 0 维就是简单的上下拼，他们的行向量内的元素数字不一样所以不能拼
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-43-98a26ca45319> in <module>

ValueError: all the input array dimensions for the concatenation axis must match
↳ exactly, but along dimension 1, the array at index 0 has size 3 and the array
↳ at index 1 has size 2
```

```
[45]: np.concatenate((array8,array10),axis = 1)
# 在第一维拼一下试试
```

```
[45]: array([[1, 2, 3, 1, 2],
            [4, 5, 6, 4, 5],
            [7, 8, 9, 7, 8]])
```

因为列向量中的元素数量一致，所以可以拼了

更加单纯的命令

使用 `np.vstack((a,b))` 去单纯的【竖着拼】，也就是第 0 维，拼【行向量】

使用 `np.hstack((a,b))` 去单纯的【横着拼】，也就是第 1 维，拼【列向量】

注意也要写双层括号，第一个括号表示传参数，第二个括号表示生成出来的是个数组

另外，此命令【不会更改】原数组

```
[50]: np.vstack((array8,array9))
# 单纯的【竖着拼】，也就是第 0 维，拼【行向量】
```

```
[50]: array([[ 1,  2,  3],
            [ 4,  5,  6],
            [ 7,  8,  9],
            [11, 21, 31],
            [41, 51, 61],
            [71, 81, 91]])
```

```
[51]: np.hstack((array8,array10))
# 单纯的【横着拼】，也就是第 1 维，拼【列向量】
```

```
[51]: array([[1, 2, 3, 1, 2],
            [4, 5, 6, 4, 5],
            [7, 8, 9, 7, 8]])
```

5.6 把矩阵拉平成向量

`array_name.flatten()` 以及 `array_name.ravel()` 去拉平矩阵成行向量

另外，此命令【不会更改】原数组

```
[53]: array8
# 看一下原矩阵
```

```
[53]: array([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])
```



```
[63]: print("使用 array8.flatten():")  
      print(array8.flatten())  
      # 拉平成行向量  
      print(" ")  
      print(array8.flatten().shape)
```

使用 `array8.flatten()`:

[1 2 3 4 5 6 7 8 9]

(9,)

```
[64]: print("使用 array8.ravel():")  
      print(array8.ravel())  
      # 拉平成行向量  
      print(" ")  
      print(array8.ravel().shape)
```

使用 `array8.ravel()`:

[1 2 3 4 5 6 7 8 9]

(9,)

6. 直接生成【行】向量 `np.r_[a:b:c]`，从 a 开始到 b（不包含 b）间隔为 c 的生成行向量

7. 直接生成【列】向量 `np.c_[a:b:c]`，从 a 开始到 b（不包含 b）间隔为 c 的生成列向量

8. 生成一个元素全是 0 的零向量/矩阵：

生成向量：`np.zeros(x, 参数)`，当中的 x 代表向量中元素个数

生成矩阵：`np.zeros([x,y], 参数)`，意为 x 行 y 列

9. 生成一个元素全是 1 的向量/矩阵：

生成向量：`np.ones(x, 参数)`，当中的 x 代表向量中元素个数

生成矩阵：`np.ones([x,y], 参数)`，意为 x 行 y 列

10. 生成一个未初期化的空向量/矩阵：

（注：这里的‘空’指的是未经过初期化，里面的值是任意的）

生成向量：`np.empty(x, 参数)`，当中的 x 代表向量中元素个数

生成矩阵：`np.empty([x,y], 参数)`，意为 x 行 y 列

这个命令存在的意义是，如果没有特别需求的话，用这个生成一个未初期化的数组速度会比以上快很多

11. 生成一个与 `array_name` 形状（维度）一样的向量/矩阵

生成一个全是 0 的：`np.zeros_like(array_name, 参数)`

生成一个全是 1 的：`np.ones_like(array_name, 参数)`

12. 生成一个 `n*n` 的单位阵（单位行列）：

`np.identity(n, 参数)`

以上所有的写着‘参数’的地方可写：`dtype = xx`，来指定元素的数据类型，一般写 `np.int32/64` 呀，`np.float32/64` 等等，默认是 `int64`

```
[19]: array1 = np.arange(1,10,2)
      # 从 1 到 10 (不包涵 10) 间隔 2 来生成数组
      print(array1)
      print(" ")
      print(array1.dtype)
      ## 没写参数的话就是默认 int64
```

[1 3 5 7 9]

int64

```
[24]: array2 = np.arange(5,12,2,dtype = np.float64)
      ## 写了 np.float64 的参数
      print(array2)
      print(" ")
      print(array2.dtype)
```

[5. 7. 9. 11.]

float64

```
[21]: np.linspace(0,10,5)
      # 从 0 到 10 这个区间内 (包涵 10) ‘等差地’ (完全均等) 的包涵 5 个数的数列
```

```
[21]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
[25]: np.random.rand(20)
      # 生成在【0 到 1 之间】的 20 个数 (随机)
```

```
[25]: array([0.73698902, 0.70055468, 0.43495538, 0.63004199, 0.54995137,
            0.15231357, 0.69437566, 0.12113391, 0.0979804 , 0.15703689,
            0.72725927, 0.37459264, 0.27299749, 0.97857657, 0.50620667,
            0.14088016, 0.59434203, 0.8315192 , 0.81517983, 0.35197632])
```

配合四舍五入的命令 `np.round(array_name, decimals = xx)` 就可以用于随机生成数据 (生成随机数据)

```
[23]: array_roundomdata = np.round(np.random.rand(30) *100, decimals = 3)
      array_roundomdata
```

```
[23]: array([91.805, 96.692,  1.431, 63.284, 16.144, 80.371, 76.763, 67.976,
            27.531, 25.062, 63.334, 15.964, 30.941, 58.734, 95.62 , 53.437,
            27.844, 65.548, 29.936, 27.699,  5.144,  4.895, 25.288, 52.95 ,
            58.759, 48.192, 75.727, 82.586, 56.95 , 69.746])
```

```
[34]: np.logspace(2,4,num = 3,base = 2,dtype = np.int64)
      # 从  $2^2$  开始到  $2^4$  为止, 生成 3 个以 2 为底的指数的数列 (实际上是等比数列), 类型为 int
```

```
[34]: array([ 4,  8, 16])
```

```
[40]: x,y = np.meshgrid(array1,array2)
      ## 生成网格, 具体逻辑见上面的解释
```

```
[45]: x
      # 因为 array2.size 是 4, 所以竖着复制 4 个【行向量】array1 作为 x 矩阵。
```

```
[45]: array([[1, 3, 5, 7, 9],
            [1, 3, 5, 7, 9],
            [1, 3, 5, 7, 9],
            [1, 3, 5, 7, 9]])
```

```
[44]: y
      # 因为 array1.size 是 5, 所以横着复制 5 个【列向量】array2 作为 y 矩阵。
```

```
[44]: array([[ 5.,  5.,  5.,  5.,  5.],
            [ 7.,  7.,  7.,  7.,  7.],
            [ 9.,  9.,  9.,  9.,  9.],
            [11., 11., 11., 11., 11.]])
```

```
[3]: np.r_[0:20:2]
      # 从 0 开始到 20 (不包含 20) 间隔为 2 的生成 [行] 向量
```

```
[3]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
[5]: np.c_[0:30:3]
# 从 0 开始到 30 (不包含 30) 间隔为 3 的生成 [列] 向量
```

```
[5]: array([[ 0],
          [ 3],
          [ 6],
          [ 9],
          [12],
          [15],
          [18],
          [21],
          [24],
          [27]])
```

```
[8]: np.zeros(10,dtype = np.int64)
# 生成元素个数为 10 的 0 向量
```

```
[8]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
[7]: np.ones(16)
# 生成元素个数为 16 的单位向量
```

```
[7]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
[9]: np.zeros([3,3],dtype = np.int64)
# 生成一个 3*3 的零矩阵
```

```
[9]: array([[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]])
```

```
[10]: np.ones([4,4],dtype = np.float32)
# 生成一个 4*4 的元素全是 1 的矩阵
```

```
[10]: array([[1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.]], dtype=float32)
```

6.0.1 灵活应用:

想构造一个全是 2 的矩阵:

```
[11]: np.ones([5,5],dtype = np.int32) * 2
```

```
[11]: array([[2, 2, 2, 2, 2],
            [2, 2, 2, 2, 2],
            [2, 2, 2, 2, 2],
            [2, 2, 2, 2, 2],
            [2, 2, 2, 2, 2]], dtype=int32)
```

构造一个全是 8 的矩阵:

```
[12]: np.ones([6,6],dtype = np.int32) * 8
```

```
[12]: array([[8, 8, 8, 8, 8, 8],
            [8, 8, 8, 8, 8, 8],
            [8, 8, 8, 8, 8, 8],
            [8, 8, 8, 8, 8, 8],
            [8, 8, 8, 8, 8, 8],
            [8, 8, 8, 8, 8, 8]], dtype=int32)
```

```
[14]: np.empty(10,dtype = np.int64)
# 生成一个未经初期化的元素个数为 10 的 ‘空’ 向量
```

```
[14]: array([
           0,           8,   140230372838968,
           1, 3275354340909339414,   139639678597460,
          105553157308416,   23157141905163276,   105553157357824,
           41])
```

```
[16]: np.empty([3,3],dtype = np.int64)
# 生成一个未经初期化的 3*3 的 ‘空’ 矩阵
```

```
[16]: array([[ 4607825790175356063, -4611042647052049244,  4603322190547985582],
            [-4611042647052049245,  4617283349392834113, -4613744806828471527],
            [ 4603322190547985580, -4613744806828471528,  4606153024599475603]])
```

```
[21]: array1 = np.array([[5,5],[6,6],[7,7],[8,8]],dtype = np.int64)
      # 生成一个 4*2 的矩阵
      print(array1)
      print(np.zeros_like(array1,dtype = np.float32))
      # 以 array1 为母版生成的全是 0 的矩阵
      print(np.ones_like(array1,dtype = np.int32))
      # 以 array1 为母版生成全是 1 的矩阵
      print(array1)
```

原数组为：

```
[[5 5]
 [6 6]
 [7 7]
 [8 8]]
```

用 `np.zeros_like` 生成的矩阵为：

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

用 `np.ones_like` 生成的矩阵为：

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

但是原矩阵 `array1` 并没有受到改变：

```
[[5 5]
 [6 6]
 [7 7]
 [8 8]]
```



```
[23]: np.identity(4, dtype = np.int32)  
      # 生成一个 4*4 的单位阵
```

```
[23]: array([[1, 0, 0, 0],  
            [0, 1, 0, 0],  
            [0, 0, 1, 0],  
            [0, 0, 0, 1]], dtype=int32)
```

7 运算

7.1 一般运算

以下**所有命令**想用来生成数组的话，得用赋值的方式来新建一个数组才行

1. 数组的对应位置的元素相乘（与矩阵形质的运算无关）：

`np.multiply(array1,array2)` 以及 `array1 * array2`

(前提是这俩必须是 Numpy 的 ndarray 类型, 且 shape 值（维度）一样)

2. 矩阵形质的乘法（线代乘法）：

`np.dot(array1,array2)`

3. 判断两个数组/矩阵的里面的各个对应位置元素的值是不是一样：

`array1 == array2`, 输出的是一个布尔值的数组

(前提是这俩必须是 ndarray 类型, 且 shape 值（维度一样）)

4. 布尔运算，逻辑运算（布尔代数）：(p.s. 输出的当然是布尔值了)

AND 运算： `np.logical_and(array1,array2)`

OR 运算： `np.logical_or(array1,array2)`

NOT 运算： `np.logical_not(array1,array2)`, 也可以只单独对一个数组进行 not 运算

5. 可以将数组的指定维度的所有元素加起来求和：

`np.sum(array_name, 参数)` 以及 `array_name.num(参数)`

参数可写： `axis = xx`, 用于指定维度，如果不写参数则默认为忽略维度地将数组中所有元素加起来，维度计数从第 0 维开始

6. 累乘（所选维度的所有元素之积）：

`array_name.prod(参数)` 以及 `np.prod(array_name, 参数)`

参数可写： `axis = xx`, 用于指定维度，如果不写参数则默认为忽略维度地将数组中所有元素乘起来

```
[41]: array1 = np.array([1,2,3])
      array2 = np.array([3,4,5])
      array3 = np.array([[1,3,5],[2,4,6],[7,8,9]])
      array4 = np.array([[2,5,7],[1,3,8],[8,8,8]])
      array5 = np.identity(3)
      # 生成一些向量和矩阵，其中 array5 是单位阵
```

```
[25]: array1 * array2
      # 简单的对应位置相乘
```

```
[25]: array([ 3,  8, 15])
```

```
[26]: np.multiply(array1,array2)
      # 简单的对应位置相乘，与前一条结果一致
```

```
[26]: array([ 3,  8, 15])
```

```
[31]: np.dot(array1,array2)
      # 线代形质的向量相乘（如果没设定向量的 shape 值，那么默认做的是行向量乘以列向量）
```

```
[31]: 26
```

```
[35]: array1.shape = 1,3
      # 设定成【行】向量 (shape 值为 (1,3)，一行三列，也就是行向量了)
      array2.shape = 3,1
      # 设定成【列】向量 (shape 值为 (3,1)，三行一列，也就是列向量)
      print("array1:",array1,array1.shape)
      print(" ")
      print("array2:")
      print(array2,array2.shape)
```

```
array1: [[1 2 3]] (1, 3)
```

```
array2:
```

```
[[3]
```

```
[4]
```

```
[5]] (3, 1)
```

```
[37]: np.dot(array1,array2)
      # 行向量乘列向量
```

```
[37]: array([[26]])
```

```
[38]: np.dot(array2,array1)
      # 列向量乘行向量
```

```
[38]: array([[ 3,  6,  9],
             [ 4,  8, 12],
             [ 5, 10, 15]])
```

```
[28]: np.dot(array3,array5)
      # 矩阵相乘, 单位阵
```

```
[28]: array([[1., 3., 5.],
             [2., 4., 6.],
             [7., 8., 9.]])
```

```
[29]: np.dot(array3,array4)
      # 矩阵相乘
```

```
[29]: array([[ 45,  54,  71],
             [ 56,  70,  94],
             [ 94, 131, 185]])
```

```
[42]: print(array1 == array2)
      print(" ")
      print(array3 == array5)
      # 看一下他们的对应位置的元素的值是不是一样的
```

```
[False False False]
```

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

```
[48]: array6 = np.array([1,1,1])
      array7 = np.array([0,0,1])
      np.logical_and(array6,array7)
      # 进行 AND 运算
```

```
[48]: array([False, False,  True])
```

```
[49]: np.logical_or(array6,array7)
      # 进行 OR 运算
```

```
[49]: array([ True,  True,  True])
```

```
[50]: np.logical_not(array6,array7)
      # 进行 not 运算
```

```
[50]: array([0, 0, 0])
```

```
[51]: array3.sum()
      # 全局求和, 忽略维度
```

```
[51]: 45
```

```
[53]: array3.sum(axis = 0)
      # 在行向量上求和, 将行向量的对应位置的元素竖着相加
```

```
[53]: array([10, 15, 20])
```

```
[54]: array3.prod()
      # 全局求累乘, 忽略维度
```

```
[54]: 362880
```

```
[56]: array3.prod(axis = 0)
      # 各行向量的对应位置元素之积。 [1*2*7, 3*4*8, 5*6*9]
```

```
[56]: array([ 14,  96, 270])
```

7.2 统计量的运算

以下**所有命令**想用来生成数组的话，得用赋值的方式来新建一个数组才行

1. 求最值：

最大值：

`array__name.max(参数)` 以及 `np.max(array__name, 参数)`

最小值：

`array__name.min(参数)` 以及 `np.min(array__name, 参数)`

参数可加 `axis = xx`

2. 找最值的索引：

最大值的索引：

`array__name.argmax(参数)` 以及 `np.argmax(array__name, 参数)`

最小值的索引：

`array__name.argmin(参数)` 以及 `np.argmin(array__name, 参数)`

参数可加： `axis = xx`

3. 求平均值：

`array__name.mean(参数)` 以及 `np.mean(array__name, 参数)`

参数可加： `axis = xx`

4. 求標準偏差 (分散の平方根)：

`array__name.std(参数)` 以及 `np.std(array__name, 参数)`

参数可加： `axis = xx`

5. 求方差 (分散)：

`array__name.var(参数)` 以及 `np.var(array__name, 参数)`

参数可加： `axis = xx`

7.2.1 最大值

```
[57]: array3
```

```
[57]: array([[1, 3, 5],  
           [2, 4, 6],  
           [7, 8, 9]])
```

```
[58]: array3.max  
      # 全局最大值
```

```
[58]: <function ndarray.max>
```

```
[59]: array3.max(axis = 0)  
      # 特定维度的最大值，这里相当于是找最大的那个行向量。
```

```
[59]: array([7, 8, 9])
```

```
[60]: array3.max(axis = 1)  
      # 相当于是找最大的那个列向量。
```

```
[60]: array([5, 6, 9])
```

```
[62]: np.max(array3)  
      # 全局最大值
```

```
[62]: 9
```

```
[63]: np.max(array3,axis = 0)  
      # 特定维度的最大值，这里相当于是找最大的那个行向量。
```

```
[63]: array([7, 8, 9])
```

```
[64]: np.max(array3,axis = 1)  
      # 相当于是找最大的那个列向量。
```

```
[64]: array([5, 6, 9])
```

7.2.2 最小值

```
[65]: array3
```

```
[65]: array([[1, 3, 5],  
           [2, 4, 6],  
           [7, 8, 9]])
```

```
[66]: array3.min()  
# 全局最小值
```

```
[66]: 1
```

```
[67]: array3.min(axis = 0)  
# 特定维度的最小值，这里相当于是找最小的那个行向量。
```

```
[67]: array([1, 3, 5])
```

```
[68]: array3.min(axis = 1)  
# 相当于是找最小的那个列向量。
```

```
[68]: array([1, 2, 7])
```

```
[69]: np.min(array3)  
# 全局最小值
```

```
[69]: 1
```

```
[70]: np.min(array3,axis = 0)  
# 特定维度的最小值，这里相当于是找最小的那个行向量。
```

```
[70]: array([1, 3, 5])
```

```
[71]: np.min(array3,axis = 1)  
# 相当于是找最小的那个列向量。
```

```
[71]: array([1, 2, 7])
```


7.2.3 找最值的索引

```
[72]: array3
```

```
[72]: array([[1, 3, 5],  
           [2, 4, 6],  
           [7, 8, 9]])
```

```
[73]: array3.argmin()  
      # 全局最小值的索引
```

```
[73]: 0
```

```
[74]: array3.argmin(axis = 1)  
      # 特性维度最小值的索引，相当于找的最小列向量的索引
```

```
[74]: array([0, 0, 0])
```

```
[75]: array3.argmin(axis = 0)  
      # 特性维度最小值的索引，相当于找的最小行向量的索引
```

```
[75]: array([0, 0, 0])
```

```
[76]: np.argmin(array3,axis = 0)  
      # 特性维度最小值的索引，相当于找的最小行向量的索引
```

```
[76]: array([0, 0, 0])
```

7.2.4 平均值

```
[78]: array3
```

```
[78]: array([[1, 3, 5],  
           [2, 4, 6],  
           [7, 8, 9]])
```

```
[80]: array3.mean() # 全局所有元素的平均值
```

```
[80]: 5.0
```

```
[81]: array3.mean(axis = 0)
```

```
[81]: array([3.33333333, 5.          , 6.66666667])
```

```
[83]: np.mean(array3,axis = 0)
```

```
[83]: array([3.33333333, 5.          , 6.66666667])
```

7.2.5 標準偏差

```
[84]: array3
```

```
[84]: array([[1, 3, 5],  
          [2, 4, 6],  
          [7, 8, 9]])
```

```
[86]: array3.std()  
# 全局所有元素的标准差
```

```
[86]: 2.581988897471611
```

```
[87]: np.std(array3)
```

```
[87]: 2.581988897471611
```

```
[88]: array3.std(axis = 0)
```

```
[88]: array([2.62466929, 2.1602469 , 1.69967317])
```

```
[89]: np.std(array3,axis = 0)
```

```
[89]: array([2.62466929, 2.1602469 , 1.69967317])
```

7.2.6 分散

```
[90]: array3
```

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

```
[91]: array3.var()
```

```
[91]: 6.666666666666667
```

```
[92]: np.var(array3)
```

```
[92]: 6.666666666666667
```

```
[93]: array3.var(axis = 0)
```

```
[93]: array([6.88888889, 4.66666667, 2.88888889])
```

```
[94]: np.var(array3,axis = 0)
```

```
[94]: array([6.88888889, 4.66666667, 2.88888889])
```

8 随机模块

以下**所有命令**想用来生成数组的话，得用赋值的方式来新建一个数组才行

需要注意的是，哪怕命令写的一样，但是**每次生成的数组中的元素都不一样**，次次都随机

本节与数组生成有重复

A. 设定生成小数点后 x 位：（影响全局的输出结果，慎用）

```
np.set_printoptions(precision = x)
```

1. 生成随机浮点数：（注：这个是默认生成的浮点数，也就是 float64）

```
np.random.rand()
```

，可以生成在【0 到 1 之间】的随机的一个数

```
np.random.rand(xx)
```

，可以生成在【0 到 1 之间】的 xx 个数（随机）

```
np.random.rand(x,y)
```

，则是用于生成 x 行 y 列的矩阵，各元素在【0 到 1 之间】

2. 生成随机整数：（注：这个是默认生成的整数，也就是 int64）

```
np.random.randint(xx)
```

，随机生成一个【0 到 xx 之间】的整数

```
np.random.randint(x,y,a)
```

，可以生成元素个数是 a 的随机行向量，元素为【x 到 y 之间】的整数（随机）

```
np.random.randint(x,size = (a,b))
```

，可以生成随机矩阵，元素为【0 到 x 之间】的整数（随机）

3. 生成服从正态分布（高斯分布，ガウス分布，正規分布）的随机数：

```
np.random.normal(mu,sigma,n)
```

，随机生成 n 个数，服从的正态分布的平均值是 mu，标准差是 sigma

还可服从其他分布，具体的去 google，或是[这个 blog\(点击这个\)](#)，这里不细致写了

4. 洗牌（打乱顺序）：

```
np.random.shuffle(array_name)
```

，将 array_name 数组中的所有元素重新随机排序，这个操作会更改原数组

B. 使得后面接下面的命令中随机生成的数固定下来，再次运行同一个命令也输出同一个结果：

```
np.random.seed(xx)
```

，其中的 xx 的位置只要有数就行，随便写什么效果都一样

如果不写里面的参数的话默认为不执行【固定结果】，也就是等于没写这个命令。

8.1 生成随机浮点数

需要注意的是，哪怕命令写的一样，但是每次生成的数组中的元素都不一样，次次都随机

```
[4]: print(np.random.rand(3,2))  
      print(np.random.rand(3,2).dtype)      # 生成随机浮点数矩阵
```

```
[[0.51628445 0.76105316]  
 [0.40554938 0.57768491]  
 [0.40193017 0.7029845 ]]
```

float64

```
[12]: np.random.rand(5)  
  
# 生成随机的浮点数的向量（元素个数 5）
```

```
[12]: array([0.40395804, 0.70343275, 0.07222818, 0.68687379, 0.94161403])
```

```
[13]: np.random.rand()      # 生成单个随机浮点数
```

```
[13]: 0.9060842752521002
```

8.2 生成随机整数

需要注意的是，哪怕命令写的一样，但是每次生成的数组中的元素都不一样，次次都随机

```
[11]: print(np.random.randint(10,size = (3,3)))  
      print(" ")  
      print(np.random.randint(10,size = (3,3)).dtype)  
      # 生成元素在【0~10之间】的随机整数矩阵
```

```
[[9 7 3]  
 [6 3 7]  
 [5 7 9]]
```

int64

```
[16]: np.random.randint(0,10,5)
      # 生成元素在【0~10之间】的随机整数行向量（元素个数 5）
```

```
[16]: array([9, 3, 8, 6, 2])
```

```
[14]: np.random.randint(10,size = (1,5))
      # 用这种一行五列的写法也是向量
      # 生成元素在【0~10之间】的随机整数行向量（元素个数 5）
```

```
[14]: array([[8, 8, 8, 2, 5]])
```

```
[15]: np.random.randint(10)      # 生成一个【0~10之间】的随机整数
```

```
[15]: 3
```

8.3 生成服从正态分布 (高斯分布, ガウス分布, 正規分布) 的随机数

需要注意的是，哪怕命令写的一样，但是每次生成的数组中的元素都不一样，次次都随机

```
[17]: np.random.normal(0,1,5)
      # 标准正态分布,  $\mu = 0, \sigma = 1$ 
```

```
[17]: array([ 0.11250964, -0.25923835, -0.20739269,  1.19142763,  0.51596814])
```

还可服从其他分布，具体的去 google
或是[这个 blog\(点击这个\)](#)
这里不细致写了

8.4 洗牌（打乱顺序）

需要注意的是，哪怕命令写的一样，但是每次生成的数组中的元素都不一样，次次都随机

```
[18]: array1 = np.arange(10)      # 生成 0-9 的 10 个数，升序
      array1
```

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

```
[19]: np.random.shuffle(array1)
      array1
```

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

8.5 乱数の初期化 (seed)

乱数を初期化するには、`random.seed()` を使います

引数を任意の数値に固定すると、毎回同じ値が生成されます

如果不写的话默认为不执行【固定结果】，也就是等于没写。

```
[24]: for i in range(3):
      np.random.seed(10)
      print(np.random.randint(0, 100))

      for i in range(3):
          np.random.seed(9) # 跟上面设定的 seed 值不一样。
          print(np.random.randint(0, 100))

      for i in range(3):
          np.random.seed(100) # 跟上面设定的两个 seed 值都不一样。
          print(np.random.randint(0, 100))
```

9

9

9

92

92

92

8

8

8

[26]: # 如果不写里面的参数的话

```
for i in range(10):  
    np.random.seed() # 没写参数  
    print(np.random.randint(0, 100))
```

92

32

69

88

6

65

88

94

78

66

9 文件读写

0. 在代码文件的同一个路径下写一个 txt

`%%writefile file_name.txt`, 按下回车后在输入想写的内容, 详见下面说明

1. 读文件进 Numpy:

这些命令 (1.1 & 1.2) 想用来生成数组的话, 得用赋值的方式来新建一个数组才行

1.1 np.loadtxt, txt 专用:

```
np.loadtxt('file_path&name', dtype='xxxxx', comments='#', delimiter=None,
skiprows=0, usecols=None, unpack=False, ndmin=0)
```

各参数说明:

file_path&name: 文件路径 (与代码文件同文件夹下就写个文件名就行了)

dtype =: 数组内的元素类型, 默认是 float

comments =: 添加 comments, 就是 python 中的 “# 这个 xxx 怎么用啥的”

delimiter =: 指定数字与数字之间的分隔符, 默认是空格

skiprows = n: 跳过后 n 行

usecols =: 使用 usecols = x 只取出第 x 【列】, 使用 usecols = (x,y,z) 仅仅只取出第 x,y,z 【列】

unpack =: 写 True 的话会将矩阵转置

ndmin = n: 返される配列の最低次元数を指定します

1.2 np.load 读取文件 (用来读 npy 和 npz 格式的):

```
np.load(file_path&name, mmap_mode=None, allow_pickle=True, fix_imports=True,
encoding='ASCII')
```

各参数说明:

file_path&name: 文件路径 (与代码文件同文件夹下就写个文件名就行了)

mmap_mode =: {None, 'r+', 'r', 'w+', 'c'} のいずれか, 默认是 None, 指定されたモード (読み込み専用か書き込み専用か両方か) でファイルを読み込む、可以不把文件内所有数据写进内存, 可以只写一部分, 适合大数据文件

allow_pickle =: 初期值 True npy ファイルとして保存されている pickle オブジェクトを読み込むかどうかを指定する

fix_imports =: 初期值 True, 用来兼容 python2 的

另外读取 npz 文件是要比 npy 文件麻烦一些的, 要一些多出来的小步骤, 在下面的实际应用例子里写了

2 从 Numpy 里把数据导出:

2.1 np.savetxt txt 文件专用:

```
np.savetxt('file_path&name', array_name, fmt='%.18e', delimiter=' ',
newline='\n',header="", footer="", comments='#')
```

各参数说明:

file_path&name: 想把文件导出的路径 (与代码文件同文件夹下就写个文件名就行了)

array_name: 导出哪个数组

delimiter =: 【列向量中】导出的文件中的数组以什么‘字符’来切分开, 建议以',' 逗号

newline =: 【行向量中】导出的文件中的数组以什么‘字符’来切分开, 建议以'\n' 换行符

fmt =: 导出时, 将数组的数据设定成到小数点的第几位, 注: 会进行四舍五入, 可写为 '%.nf' 就是小数点后第 n 位

header = 'xxx': 导出时, 在文件【开头】写入 xxx

footer = 'YYY': 导出时, 在文件【末尾】写入 YYY

comments: 在插入 header や footer 的时候、指定 header や footer 前面的字符

2.2 np.save(z) 导出文件, npy 和 npz 格式专用:

npz 格式版:

```
np.save('file_path&name',array_name,allow_pickle=True, fix_imports=True)
```

npz 格式版:

```
np.savez('file_path&name',array_name_1,array_name_2, array_name_3,
allow_pickle=True,fix_imports=True)
```

各参数说明:

file_path&name: 文件路径 (与代码文件同文件夹下就写个文件名就行了)

array_name: 导出哪个数组

allow_pickle =: 默认为 True, 指定如何设置 pickle オブジェクト (npz ファイルとして保存されている的时候)

fix_imports =: 默认为 True, 用来兼容 python2 的

具体参考 (txt): [Click this](#)

具体参考 (npz 和 npz): [Click this](#)

npz 和 npz 格式的介绍:

都可以そのまま的保存住 np.ndarray 的结构

【npz】只能用来保存一个数组

【npz】可以保存多个数组, 就像生成了一个压缩包, 解压后出来是几个 npz 格式的文件, 每个 npz 文件对应了一个数组

9.1 TXT 类型文件:

9.1.1 从外面读取进 numpy,python

```
[4]: %%writefile test.txt
1 2 3 4 5 6 7 8 9
5 6 3 2 5 5 2 6 8
# 想换行直接敲回车就行
```

Overwriting test.txt

```
[5]: np.loadtxt('test.txt')
# 很简单的就能读进来。
```

```
[5]: array([[1., 2., 3., 4., 5., 6., 7., 8., 9.],
          [5., 6., 3., 2., 5., 5., 2., 6., 8.]])
```

```
[6]: %%writefile test2.txt
1,2,3,4,5,6,7,8,9
5,6,3,2,5,5,2,6,8
```

Writing test2.txt

```
[7]: np.loadtxt('test2.txt')
# 这次直接读就不行了。
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-7-d96b86678505> in <module>

ValueError: could not convert string to float: '1,2,3,4,5,6,7,8,9'
```

因为带了逗号，所以不加一些参数的直接读，会导致报错，因为逗号不是数字

```
[8]: np.loadtxt('test2.txt', delimiter = ',')
# 加了参数，把逗号','作为分隔符，就不报错了。
```

```
[8]: array([[1., 2., 3., 4., 5., 6., 7., 8., 9.],
          [5., 6., 3., 2., 5., 5., 2., 6., 8.]])
```

```
[9]: %%writefile test3.txt
x,y,e,w,w,y,u,s,a
1,2,3,4,5,6,7,8,9
5,6,3,2,5,5,2,6,8
# 创建一个新的测试文件，跟上面比多加了一行纯文字
```

Writing test3.txt

```
[10]: np.loadtxt('test3.txt',delimiter = ',')
# 这次直接读还是不行。因为第一行也不是数字
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-10-af28bfaf24d7> in <module>

ValueError: could not convert string to float: 'x'
```

```
[11]: np.loadtxt('test3.txt',delimiter = ',',skiprows = 1)
# 这次把第一行跳过去，然后指定逗号 ‘,’ 为分隔符
```

```
[11]: array([[1., 2., 3., 4., 5., 6., 7., 8., 9.],
            [5., 6., 3., 2., 5., 5., 2., 6., 8.]])
```

```
[2]: np.loadtxt('test3.txt',delimiter = ',',skiprows = 1,usecols = (0,1,4))
# 使用 usecols = x 只取出第 x 【列】，使用 usecols = (x,y,z) 仅仅只取出第 x,y,z 【列】
```

```
[2]: array([[1., 2., 5.],
            [5., 6., 5.]])
```

9.1.2 从 Numpy 里导出:

```
[24]: array_out = np.random.rand(5,5)
array_out
# 随机生成一个浮点数数组用来导出着玩
```

```
[24]: array([[0.39628657, 0.11774223, 0.73531755, 0.81650139, 0.26460353],
            [0.29261334, 0.29556388, 0.00660806, 0.60178655, 0.33365383],
            [0.16130804, 0.33707648, 0.37549226, 0.41077773, 0.35719691],
            [0.19115583, 0.46328589, 0.32768576, 0.28775594, 0.64830785],
            [0.30946639, 0.37215186, 0.98424042, 0.88911226, 0.23382535]])
```

```
[29]: np.savetxt('test_out.txt',array_out,delimiter = ',',newline = '\n',fmt = '%.
      ↪2f',header = '这个是开头',footer = '这个是末尾',comments = '>>>')
# 将 array_out 输出到代码文件同文件夹，横着的两个数用 ‘,’ 相隔开，竖着的行与行之间的
  区分方式为换行 '\n'，设定为小数点后两位，开头文字 & 末尾文字的前面写个 '>>>'
```

9.2 npy 和 npz 类型文件:

9.2.1 从 Numpy 里导出:

```
[8]: array_test1 = np.loadtxt('test.txt')
array_test1
# 借助上面的命令来生成数组
```

```
[8]: array([[1., 2., 3., 4., 5., 6., 7., 8., 9.],
           [5., 6., 3., 2., 5., 5., 2., 6., 8.]])
```

```
[21]: np.save('test.npy',array_test1)
# 导出成 npz 格式, 保留 ndarray 的形式
```

```
[31]: array_test2 = np.array([[5,5],[6,6],[7,7],[8,8]],dtype = np.int64)
# 多加一个数组
array_test2
```

```
[31]: array([[5, 5],
           [6, 6],
           [7, 7],
           [8, 8]])
```

```
[39]: np.savez('test.npz',a = array_test1,b=array_test2)
# 把两个数组导出成 npz 格式, 保留 ndarray 的形式, 并且在导出的文件中重命名了一下数组
```

9.2.2 从外面读取进 numpy,python

```
[24]: np.load('test.npy')
# 直接读取进来, 很方便
```

```
[24]: array([[1., 2., 3., 4., 5., 6., 7., 8., 9.],
           [5., 6., 3., 2., 5., 5., 2., 6., 8.]])
```

```
[34]: test_data = np.load('test.npz')
# 生成一个 test_data 来容纳读取进来的 npz 文件, 因为 npz 文件一个压缩包/集合包
# 里面包含了多个 npz 文件
```

```
[5]: list(test_data.keys())  
# 用于查看 test_data 里面容纳了什么
```

```
[5]: ['a', 'b']
```

```
[30]: test_data['a']
```

```
[30]: array([[1., 2., 3., 4., 5., 6., 7., 8., 9.],  
           [5., 6., 3., 2., 5., 5., 2., 6., 8.]])
```

```
[37]: test_data['b']
```

```
[37]: array([[5, 5],  
           [6, 6],  
           [7, 7],  
           [8, 8]])
```

```
[43]: np.savez('test_none_array_name_set.npz', array_test1, array_test2)  
# 在储存的时候也可以不指定在保存文件中 array 被重新存成什么名字
```

```
[44]: test_data2 = np.load('test_none_array_name_set.npz')  
# 读进来看一下
```

```
[46]: list(test_data2.keys())  
# 如果导出文件的中的名字没有被指定的话，读出来的数组的名字就会被写成下面哪种的“默认名字”
```

```
[46]: ['arr_0', 'arr_1']
```

10 Numpy 练习题

打印当前 Numpy 版本

```
[51]: print(np.__version__)  
# 注意前后是双下划线 __ , 不是一个双下划线
```

1.20.1

构造一个全零的矩阵, 并打印其占用的内存大小

```
[61]: array1 = np.zeros((5,5))  
print("%d Byte"%(array1.size * array1.itemsize))
```

200 Byte

打印一个函数的帮助文档, 比如 numpy.add

```
[ ]: print(help(np.info(np.add)))  
# 结果太长就不执行了
```

创建一个 10-49 的数组, 并将其倒序排列

```
[79]: array2 = np.arange(10,50,1)  
array2[::-1]
```

```
[79]: array([49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33,  
          32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16,  
          15, 14, 13, 12, 11, 10])
```

找到一个数组中不为 0 的索引

```
[92]: np.random.seed(10)  
array3 = np.random.randint(0,2,15)  
np.where(array3 > 0)
```

```
[92]: (array([ 0,  1,  3,  5,  6,  8,  9, 11, 12]),)
```

```
[94]: np.nonzero(array3)  
# 方法二
```

```
[94]: (array([ 0,  1,  3,  5,  6,  8,  9, 11, 12]),)
```


随机构造一个 3*3 矩阵，并打印其中最大与最小值

```
[102]: np.random.seed(10)
        array4 = np.random.randint(0,20,(3,3))
        print(array4)
        print("最大值为: ",np.max(array4))
        print("最小值为: ",np.min(array4))
```

```
[[ 9  4 15]
 [ 0 17 16]
 [17  8  9]]
最大值为: 17
最小值为: 0
```

构造一个 5*5 的矩阵，令其值都为 1，并在最外层加上一圈 0

```
[116]: array5 = np.zeros((7,7),dtype = np.int64)
array5[1:6,1:6] = np.ones((5,5),dtype = np.int64)
array5 # 方法一
```

[illegible]

```
[119]: array6 = np.ones((5,5),dtype = np.int64)
array6 = np.pad(array6,pad_width=1,mode='constant',constant_values=0)
array6 # 方法二
```

[illegible]

构建一个 shape 为 (6, 7, 8) 的矩阵, 并找到第 100 个元素的索引值

```
[2]: np.unravel_index(100,(6,7,8))
```

```
[2]: (1, 5, 4)
```

对一个 5*5 的矩阵做归一化操作

```
[6]: array = np.random.randint(0,100,(5,5))
array_max = array.max()
array_min = array.min()
array = (array - array_min) / (array_max - array_min)
array
```

```
[6]: array([[0.93333333, 0.73333333, 0.83333333, 0.67777778, 0.          ],
           [0.67777778, 0.43333333, 0.9          , 1.          , 0.43333333],
           [0.73333333, 0.46666667, 0.16666667, 0.72222222, 0.3          ],
           [0.66666667, 0.26666667, 0.43333333, 0.5          , 0.36666667],
           [0.66666667, 0.16666667, 0.82222222, 0.76666667, 0.65555556]])
```

找到两个数组中相同的值

```
[8]: z1 = np.random.randint(0,10,10)
z2 = np.random.randint(0,10,10)
print(z1)
print(" ")
print(z2)
print(" ")
print(np.intersect1d(z1,z2))
# 注意: 是 intersect1d, d 前面是数字 '1', 不是字母 'l'
```

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

```
[6 3 0 9 0 6 2 8 2 4]
```

```
[0 2 3 4 8]
```

得到今天明天昨天的日期

```
[11]: today = np.datetime64('today','D')
      yesterday = np.datetime64('today','D') - np.timedelta64(1,'D')
      tomorrow = np.datetime64('today','D') + np.timedelta64(1,'D')
      print(today)
      print(yesterday)
      print(tomorrow)
```

2022-02-20

2022-02-19

2022-02-21

得到一个月所有日期

```
[12]: np.arange('2022-02','2022-03',dtype = 'datetime64[D]')
```

```
[12]: array(['2022-02-01', '2022-02-02', '2022-02-03', '2022-02-04',
            '2022-02-05', '2022-02-06', '2022-02-07', '2022-02-08',
            '2022-02-09', '2022-02-10', '2022-02-11', '2022-02-12',
            '2022-02-13', '2022-02-14', '2022-02-15', '2022-02-16',
            '2022-02-17', '2022-02-18', '2022-02-19', '2022-02-20',
            '2022-02-21', '2022-02-22', '2022-02-23', '2022-02-24',
            '2022-02-25', '2022-02-26', '2022-02-27', '2022-02-28'],
            dtype='datetime64[D]')
```

得到一个数的整数部分

(是要取整数部分，不是要四舍五入)

```
[19]: z = np.random.uniform(0,10,10)
      ## 从均匀分布中取数，具体用法参考从正态分布中取数，用法一样
      print(z)
      print(np.floor(z))
```

```
[9.50658965 1.42431977 6.579335 5.43848878 5.57160479 0.37432711
 2.3409184 8.05329838 1.87695786 1.75665883]
```

```
[9. 1. 6. 5. 5. 0. 2. 8. 1. 1.]
```

打印数组的部分值 (省略一部分), 所有值

```
[34]: np.set_printoptions(threshold = 5)
      #threshold = x , 这个 x 设置成任何整数结果都一样
      z = np.zeros((15,15))
      z
```

```
[34]: array([[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.]])
```

```
[40]: np.set_printoptions(threshold=np.inf)
      #threshold = np.inf , 就是打印所有值了
      y = np.zeros((15,15))
      y
```

[illegible]

构造一个数组让它不能被改变

read-only

```
[22]: z = np.zeros((5,5))
      z.flags.writeable = False
      z[0] = 1
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-22-c8aff6376c95> in <module>
----> 3 z[0] = 1

ValueError: assignment destination is read-only
```

找到在一个数组中，最接近一个数的索引

```
[46]: z = np.arange(100)
      v = np.random.uniform(0,100)
      print(v)
      index = (np.abs(z-v)).argmin()
      # 取绝对值 np.abs ()
      # 上面的意思是，让 z-v，得到一个数组，然后取这个数组的绝对值
      # 那个数组中的所有元素都变成了原来元素的绝对值，然后再找这个被‘绝对值化’的数组中的
      # 最小值的索引
      print(z[index])
      # 最后再把上面求的索引代入回 z
```

25.059063249877266

25

32 位 float 类型和 32 位 int 类型的转换

```
[47]: z = np.arange(10, dtype = np.float32)
      print(z.dtype)
      z = z.astype(np.int32)
      print(z.dtype)
```

float32

int32

打印数组元素位置坐标与数值

```
[50]: z = np.arange(9).reshape(3,3)
      for index,value in np.ndenumerate(z):
          #np.ndenumerate() 就是单纯的用于枚举矩阵的 index 和对应位置元素值的
          #单独拿出去用似乎没啥用，就得这么配合这个固定的算法似乎
          print(index,value)
```

```
(0, 0) 0
(0, 1) 1
(0, 2) 2
(1, 0) 3
(1, 1) 4
(1, 2) 5
(2, 0) 6
(2, 1) 7
(2, 2) 8
```

按照数组的某一列进行排序

```
[86]: z = np.random.randint(0,10,(3,3))
      print(z)
      print(z[z[:,0].argsort()])
      # 按照第【0】列来排序，升序
      print(z[z[:,1].argsort()])
      # 按照第【1】列来排序，升序
```

```
[[4 3 9]
 [4 0 4]
 [7 1 3]]
```

```
[[4 3 9]
 [4 0 4]
 [7 1 3]]
```

```
[[4 0 4]
 [7 1 3]
 [4 3 9]]
```

统计数组中每个数值出现的次数

```
[96]: z = np.random.randint(0,15,12)
print(z)
print(" ")
print(np.bincount(z))
#np.bincount 意为: 从 0 开始到 array.max (数组中最大值), 数字都各出现了几次
# 哪怕数组中没有 0 都会从 0 开始统计
```

```
[11  2  9  0  8  4 13  0  8  5  1  7]
```

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

对四维数组最后的两维进行求和

```
[104]: z = np.random.randint(0,10,(2,2,2,2))
res = z.sum(axis = (-2,-1))
print(z)
print(" ")
print(res)
```

```
[[[1 1]
  [1 9]]
```

```
[[2 0]
 [6 4]]]
```

```
[[[1 4]
  [9 0]]
```

```
[[4 1]
 [0 4]]]
```

```
[[12 12]
 [14  9]]
```

交换矩阵当中的两行

```
[107]: z = np.arange(25).reshape(5,5)
print(z)
print(" ")
z[[0,1]] = z[[1,0]]
print(z)
```

```
[[ 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]]
```

```
[[ 5  6  7  8  9]
 [ 0  1  2  3  4]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

找到数组中最常出现的数字

```
[109]: z = np.random.randint(0,10,11)
print(z)
print(" ")
print(np.bincount(z).argmax())
```

```
[9 1 5 9 2 5 8 0 0 6 7]
```

0

快速查找数组中最大的 n 个数

```
[110]: z = np.arange(100000)
np.random.shuffle(z)
# 先随机洗牌一下，不然 arange 出来的数都是从 0 开始升序排的
n = 5
print(z[np.argpartition(-z,n)[:n]])
```

```
[99999 99998 99997 99996 99995]
```


去掉一个数组中的所有“元素都相同”的数据

```
[115]: a = np.array([1,2,3,4,5])
      b = np.array([1,2,3,4,4])

      np.all(a == b)
      # 必须两个数组中所有值都相等才会输出 true
```

[115]: False

```
[116]: np.any(a == b)
      # 只要有一个值相等就输出 true
```

[116]: True

```
[150]: z = np.random.randint(0,2,(10,3))
      z
```

```
[150]: array([[1, 1, 1],
              [0, 0, 1],
              [1, 0, 0],
              [0, 0, 0],
              [0, 0, 0],
              [1, 1, 0],
              [0, 0, 0],
              [0, 0, 0],
              [0, 0, 1],
              [1, 0, 0]])
```

```
[152]: e = np.all(z[:,1:] == z[:, :-1], axis = 1)
      e
```

```
[152]: array([ True, False, False,  True,  True, False,  True,  True, False,
              False])
```

```
[156]: ~e
      # 在逻辑数组前加~, 表示将整个数组做 not 运算
```

```
[156]: array([False,  True,  True, False, False,  True, False, False,  True,
              True])
```

```
[155]: z[~e]  
# 再把 not(e) 传进 z, 这样, 就去掉了 ‘元素都一样’ 的行
```

```
[155]: array([[0, 0, 1],  
              [1, 0, 0],  
              [1, 1, 0],  
              [0, 0, 1],  
              [1, 0, 0]])
```

参考文献

- [1] Wes McKinney(2012) 『Python for Data Analysis』 ISBN:9781449319793
- [2] BiliBili-人工智能精品教程库 (2022) 『【人工智能必备：Python 数据分析】浙大博士半天就教会我大学一直没学会的利用 Python 进行数据分析！怎么可以讲的如此通俗，太强了！』
<https://www.bilibili.com/video/BV1ru411U772>
- [3] Hbk project(2020) 『Python ランダムな数値の生成方法 (random モジュールの使い方) 』
https://hibiki-press.tech/python/pseudo_random_number/1431
- [4] 知乎-AyiStar(2020) 『Python 切片完全指南 (语法篇)』<https://zhuanlan.zhihu.com/p/79541418>
- [5] Spot Inc.(2017) 『np.load と np.save 関数でファイルの永続化方法』
<https://deepage.net/features/numpy-loadsave.html>
- [6] Spot Inc.(2017) 『np.loadtxt と np.savetxt でテキストファイルを読み書きする方法』
<https://deepage.net/features/numpy-loadsavetxt.html>