目　录

引入

0.1 版本需求

本视频中，使用的 Python 解释器与第三方库的版本如下。

- Python 为 3.9 版本，自 3.4 以来改动的语法可忽略不计，除非更新到 4.0。
- NumPy 为 1.21 版本，自发行以来改动的语法可忽略不计，除非更新到 2.0，不同版本的发行日志：https://numpy.org/devdocs/release/1.25.0-notes.html。

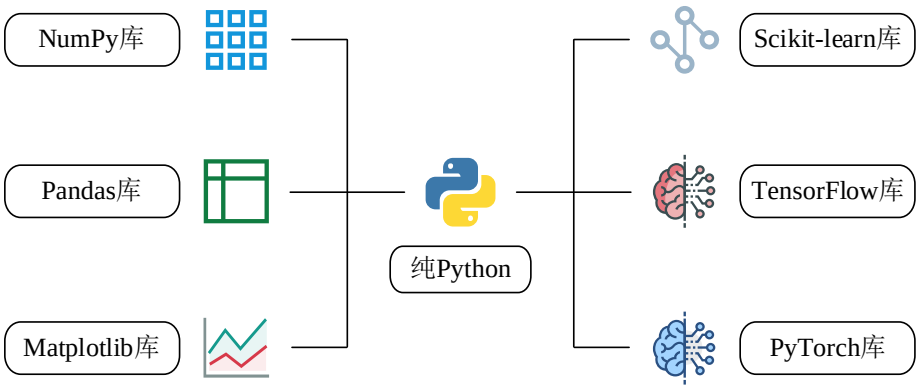
0.2 视频特点

- 本视频分辨率为 1080P，请调高分辨率；
- 视频简介与置顶评论中均附有讲义链接，此讲义为原创，请勿商用。

0.3 视频 UP 主

- UP 的本科为三峡大学（原电力部 6 所直属高校之一，超强电气型），硕士是中南大学（计算机、自动化、临床、护理等热门专业均属 A 类学科）。
- 如果课件中有纰漏，请在视频评论区反馈。

0.4 深度学习的相关库



① NumPy 包为 Python 加上了关键的数组变量类型，弥补了 Python 的不足；
② Pandas 包在 NumPy 数组的基础上添加了与 Excel 类似的行列标签；
③ Matplotlib 库借鉴 Matlab，帮 Python 具备了绘图能力，使其如虎添翼；
④ Scikit-learn 库是机器学习库，内含分类、回归、聚类、降维等多种算法；
⑤ TensorFlow 库是 Google 公司开发的深度学习框架，于 2015 年问世；
⑥ PyTorch 库是 Facebook 公司开发的深度学习框架，于 2017 年问世。

0.5 深度学习的基本常识

- **人工智能**是一个很大的概念，其中一个最重要的分支就是机器学习；
- **机器学习**的算法多种多样，其中最核心的就是神经网络；
- **神经网络**的隐藏层若足够深，就被称为深层神经网络，也即深度学习；
- **深度学习**包含深度神经网络、卷积神经网络、循环神经网络等。

一、数组基础

导入 NumPy 时，通常给其一个别名“np”，即 **import** numpy **as** np。

NumPy 库中的函数，要在函数名前加上导入的库名 np. 才能使用。

1.1 数据类型

（1）整数型数组与浮点型数组

为克服列表的缺点，一个 NumPy 数组只容纳一种数据类型，以节约内存。为方便起见，可将 NumPy 数组简单分为整数型数组与浮点型数组。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建整数型数组
arr1 = np.array( [1, 2, 3] )          # 元素若都是整数，则为整数型数组
print(arr1)
```
[1 2 3]

In [3] :
```
# 创建浮点型数组
arr2 = np.array( [1.0, 2, 3] )      # 内含浮点数，则为浮点型数组
print(arr2)
```
[1. 2. 3.]

注意，使用 print 输出 NumPy 数组后，元素之间没有逗号，这有两个好处，一是可以可将之与 Python 列表区分开来，二是避免逗号与小数点之间的混淆。

（2）同化定理

一个人的力量是无法改变全体的，在实际操作中要注意：

- 往整数型数组里插入浮点数，该浮点数会自动被截断为整数；
- 往浮点型数组里插入整数，该整数会自动升级为浮点数；

In [1] :
```
import numpy as np
```

In [2] :
```
# 整数型数组
arr1 = np.array( [1, 2, 3] )
arr1[0] = 100.9                  # 插入浮点数，被截断，数组仍为整数型
print(arr1)
```
[100 2 3]

In [3] :
```
# 浮点型数组
arr2 = np.array( [1.0, 2, 3] )
arr2[1] = 10                     # 插入整数型，被升级，数组仍为浮点型
print(arr2)
```
[1. 10. 3.]

（3）共同改变定理

同化定理告诉我们，整数型数组和浮点型数组之间的界限十分严格，那么如何将这两种数据类型的数组进行相互转化呢？既然某一个人容易被集体所同化，那只要全体共同改变，自然就可以成功。

整数型数组和浮点型数组相互转换，规范的方法是使用 .astype( ) 方法。

In [1] :
```
import numpy as np
```

In [2] :
```
# 整数型数组
arr1 = np.array( [1, 2, 3] )
print(arr1)
```
```
[1 2 3]
```

In [3] :
```
# 整数型数组 ——> 浮点型数组
arr2 = arr1.astype(float)
print(arr2)
```
```
[1. 2. 3.]
```

In [4] :
```
# 浮点型数组 ——> 整数型数组
arr3 = arr2.astype(int)
print(arr3)
```
```
[1 2 3]
```

除了上述方法，只要满足共同改变定理，整数型数组和浮点型数组仍然可以互相转换。最常见的是整数型数组在运算过程中升级为浮点型数组，示例如下。

In [1] :
```
import numpy as np
```

In [2] :
```
# 整数型数组
arr = np.array( [1, 2, 3] )
print(arr)
```
```
[1 2 3]
```

In [3] :
```
# 整数型数组与浮点数做运算
print( arr + 0.0 )
print( arr * 1.0 )
```
```
[1. 2. 3.]
[1. 2. 3.]
```

In [4] :
```
# 整数型数组遇到除法（即便是除以整数）
print( arr / 1 )
```
```
[1. 2. 3.]
```

In [5] :
```
# 整数型数组与浮点型数组做运算
int_arr = np.array( [1, 2, 3] )
float_arr = np.array( [1.0, 2, 3] )
print( int_arr + float_arr )
```
```
[2. 4. 6.]
```

整数型数组很好升级，但浮点型数组在运算过程中一般不会降级。

1.2 数组维度

（1）一维数组与二维数组

考虑到深度学习中三维及其以上的数组出现次数少，我们后续主要讲解 NumPy 中的一维数组和二维数组，学了一维和二维后，很好类推到三维。

- 不同维度的数组之间，从外形上的本质区别是

  一维数组使用 1 层中括号表示；

  二维数组使用 2 层中括号表示；

  三维数组使用 3 层中括号表示。

- 有些函数需要传入数组的形状参数，不同维度数组的形状参数为

  一维数组的形状参数形如：`x` 或 `(x,)`；

  二维数组的形状参数形如：`(x,y)`；

  三维数组的形状参数形如：`(x,y,z)`。

- 现在以同一个序列进行举例

  当数组有 1 层中括号，如[1 2 3]，则其为一维数组，其形状是 `3` 或 `(3,)`；

  当数组有 2 层中括号，如[[1 2 3]]，则其为二维数组，其形状是 `(1,3)`；

  当数组有 3 层中括号，如[[[1 2 3]]]，则其为三维数组，其形状是 `(1,1,3)`；

这里用后面要讲的 np.ones() 函数进行演示，只因其刚好需要传入形状参数。

In [1]:
```
import numpy as np
```

In [2]:
```
arr1 = np.ones(3)       # 传入形状 3
print(arr1)             # 造出一维数组
```
```
[1. 1. 1.]
```

In [3]:
```
arr2 = np.ones((1,3))       # 传入形状(1,3)
print(arr2)                 # 造出二维数组
```
```
[[1. 1. 1.]]
```

In [4]:
```
arr3 = np.ones((1,1,3))       # 传入形状(1,1,3)
print(arr3)                   # 造出三维数组
```
```
[[[1. 1. 1.]]]
```

同时，我们还可以使用数组的 .shape 属性查看 arr1 和 arr2 的形状。

In [5]:
```
print( arr1.shape )
```
```
(3,)
```

In [6]:
```
print( arr2.shape )
```
```
(1, 3)
```

In [7]:
```
print( arr3.shape )
```
```
(1, 1, 3)
```

大家可以随时留意一下数组的维度（通过中括号的数量），后面有些函数（比如数组的拼接函数）需要两个数组是同维度的。

（2）不同维度数组之间的转换

一维数组转二维数组，还是二维数组转一维数组，均要使用的是数组的重塑方法 .reshape( ) ，该方法需要传入重塑后的形状（shape）参数。

这个方法神奇的是，给定了其他维度的数值，剩下一个维度可以填-1，让它自己去计算。比如把一个 5 行 6 列的矩阵重塑为 3 行 10 列的矩阵，当列的参数 10 告诉它，行的参数直接可以用-1 来替代，它会自己去用 30 除以 10 来计算。

首先，演示将一维数组升级为二维数组。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建一维数组
arr1 = np.arange(10)
print(arr1)
```
```
[0 1 2 3 4 5 6 7 8 9]
```

In [3] :
```
# 升级为二维数组
arr2 = arr1.reshape( (1,-1) )
print(arr2)
```
```
[[0 1 2 3 4 5 6 7 8 9]]
```

这里给.reshape( )传入的形状参数是(1,-1)，正常情况下我们都肯定是(1,10)，这里(1,-1)的含义是：行的参数是 1 行，列的参数-1 自己去除着算吧。

接着，演示将二维数组降级为一维数组。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建二维数组
arr2 = np.arange(10).reshape(2,5)
print(arr2)
```
```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

In [3] :
```
# 降级为一维数组
arr1 = arr2.reshape( -1 )
print(arr1)
```
```
[0 1 2 3 4 5 6 7 8 9]
```

这里给.reshape( )传入的形状参数是 -1 ，正常情况应该是 10，这里-1 的含义是：反正是一维数组，形状直接自己去算吧。

现规定，本讲义中，将一维数组称为向量，二维数组称为矩阵。

二、数组的创建

2.1 创建指定数组

当明确知道数组每一个元素的具体数值时，可以使用 np.array( ) 函数，将 Python 列表转化为 NumPy 数组。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建一维数组——向量
arr1 = np.array( [1,2,3] )
print(arr1)
```
[1 2 3]

In [3] :
```
# 创建二维数组——行矩阵
arr2 = np.array( [ [1,2,3] ] )
print(arr2)
```
[[1 2 3]]

In [4] :
```
# 创建二维数组——列矩阵
arr3 = np.array( [ [1],[2],[3] ] )
print(arr3)
```
[[1]
 [2]
 [3]]

In [5] :
```
# 创建二维数组——矩阵
arr4 = np.array( [ [1,2,3],[4,5,6] ] )
print(arr4)
```
[[1 2 3]
 [4 5 6]]

2.2 创建递增数组

递增数组使用 np.arange( ) 函数进行创建（arange 全称是 array_range）。

In [1] :
```
import numpy as np
```

In [2] :
```
# 递增数组
arr1 = np.arange(10)      # 从 0 开始，到 10 之前结束
print(arr1)
```
[0 1 2 3 4 5 6 7 8 9]

In [3] :
```
# 递增数组
arr2 = np.arange(10,20)    # 从 10 开始，到 20 之前结束
print(arr2)
```
[10 11 12 13 14 15 16 17 18 19]

In [4] :
```
# 递增数组
arr3 = np.arange(1,21,2)  # 从 1 开始，到 21 之前结束，步长为 2
print(arr3)
```
[1 3 5 7 9 11 13 15 17 19]

2.3 创建同值数组

需要创建同值数组时，使用 np.zeros() 函数以及 np.ones() 函数，如示例。

In [1]:
```
import numpy as np
```

In [2]:
```
# 全 0 数组
arr1 = np.zeros( 3 )                # 形状为 3 的向量
print(arr1)
```
```
[0. 0. 0.]
```

In [3]:
```
# 全 1 数组
arr2 = np.ones( (1,3) )             # 形状为(1,3)的矩阵
print(arr2)
```
```
[[1. 1. 1.]]
```

In [4]:
```
# 全 3.14 数组
arr3 = 3.14 * np.ones( (2,3) )      # 形状为(2,3)的矩阵
print(arr3)
```
```
[[3.14 3.14 3.14]
 [3.14 3.14 3.14]]
```

示例中隐藏了一个细节——两个函数输出的并不是整数型的数组，这可能是为了避免插进去的浮点数被截断，所以将其设定为浮点型数组。

2.4 创建随机数组

有时需要创建随机数组，那么可以使用 np.random 系列函数，如示例所示。

In [1]:
```
import numpy as np
```

In [2]:
```
# 0-1 均匀分布的浮点型随机数组
arr1 = np.random.random( 5 )                # 形状为 5 的向量
print( arr1 )
```
```
[0.59699399  0.89113584  0.00695752  0.49089431  0.32050609]
```

在 In [2]中，如果想创建 60-100 范围内均匀分布的 3 行 3 列随机数组，可输入 (100-60)*np.random.random((3,3)) + 60。

In [3]:
```
# 整数型随机数组
arr2 = np.random.randint( 10,100,(1,15) )      # 形状为(1,15)的矩阵
print( arr2 )
```
```
[[17 65 54 48 82 57 52 26 28 27 53 36 61 92 13]]
```

在 In [3]中，该函数需要额外输入范围参数，本例中范围是 10-100。

In [4]:
```
# 服从正态分布的随机数组
arr3 = np.random.normal( 0,1,(2,3) )          # 形状为(2,3)的二维矩阵
print( arr3 )
```
```
[[-0.43163964 -1.56817412  0.5460523 ]
 [-2.93093358  0.42577899 -1.69842077]]
```

在 In [4]中，该函数需要额外输入正态参数，本例中均值为 0、标准差为 1，这种情况可直接使用 np.random.randn()函数，只需要传入形状参数即可。

三、数组的索引

前面我们规定，将一维数组称为向量，将二维数组称为矩阵。

3.1 访问数组元素

与 Python 列表一致，访问 NumPy 数组元素时使用中括号，索引由 0 开始。

（1）访问向量

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建向量
arr1 = np.arange( 1,10 )
print( arr1 )
```
```
[1 2 3 4 5 6 7 8 9]
```

In [3] :
```
# 访问元素
print( arr1[3] )        # 正着访问
print( arr1[-1] )       # 倒着访问
```
```
4
9
```

In [4] :
```
# 修改数组元素
arr1[3] = 100;
print( arr1 )
```
```
[1 2 3 100 5 6 7 8 9]
```

（2）访问矩阵

In [5] :
```
# 创建矩阵
arr2 = np.array( [ [1,2,3],[4,5,6] ] )
print(arr2)
```
```
[[1 2 3]
 [4 5 6]]
```

In [6] :
```
# 访问元素
print( arr2[0,2] )
print( arr2[1,-2] )
```
```
3
5
```

In [7] :
```
# 修改元素
arr2[1,1] = 100.9
print( arr2 )
```
```
[[  1   2   3]
 [  4 100   6]]
```

在 In [7]中，浮点数 100.9 插入到整数型数组时被截断了。

3.2 花式索引

花式索引（Fancy indexing）又名“花哨的索引”，UP 认为不应该用“花哨”来形容，这里的 Fancy 应取“华丽的、巧妙的、奢华的、时髦的”之义。

上一小节访问单个元素时，向量用 arr1[x]，矩阵用 arr2[x,y]。逗号在矩阵里用于区分行与列，这一小节，逗号新增一个功能，且不会与矩阵里的逗号混淆。**普通索引用一层中括号，花式索引用两层中括号。**

（1）向量的花式索引

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建向量
arr1 = np.arange(0,90,10)
print(arr1)
```
```
[ 0 10 20 30 40 50 60 70 80]
```

In [3] :
```
# 花式索引
print( arr1[ [0,2] ] )
```
```
[0 20]
```

（2）矩阵的花式索引

In [4] :
```
# 创建矩阵
arr2 = np.arange(1,17).reshape(4,4)
print(arr2)
```
```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

In [5] :
```
# 花式索引
print( arr2[ [0,1] , [0,1] ] )
print( arr2[ [0,1,2] , [2,1,0] ] )
```
```
[1 6]
[3 6 9]
```

In [6] :
```
# 修改数组元素
arr2[ [0,1,2,3] , [3,2,1,0] ] = 100
print(arr2)
```
```
[[  1   2   3 100]
 [  5   6 100   8]
 [  9 100  11  12]
 [100  14  15  16]]
```

根据以上实例，花式索引输出的仍然是一个向量。

表 3-1 普通索引与花式索引的区别

索引方式	向量	矩阵
普通索引	arr1[x_1]	arr2[x_1 , y_1]
花式索引	arr1[[x_1,x_2,…,x_n]]	arr2[[x_1,x_2,…,x_n] , [y_1,y_2,…,y_n]]

3.3 访问数组切片

（1）向量的切片

向量与列表切片的操作完全一致，因此本页内容在 Python 基础中均有涉及。

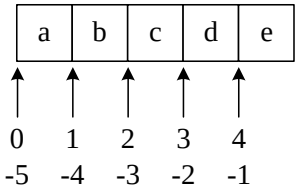


图 3-1 索引负责其指向区域的右侧一个单元格

In [1]:
```
import numpy as np
arr1 = np.arange(10)
```

当明确知道从第 x 个元素切到第 y 个元素，如示例所示。

In [2]:
```
print(arr1)
print(arr1[1:4])     # 从索引[1]开始，切到索引[4]之前
print(arr1[1:  ])    # 从索引[1]开始，切到结尾
print(arr1[  :4])    # 从数组开头开始，切到索引[4]之前
```

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

当明确切除数组的开头与结尾，如示例所示。

In [3]:
```
print(arr1)
print(arr1[2:-2])     # 切除开头 2 个和结尾 2 个
print(arr1[2:  ])     # 切除开头 2 个
print(arr1[  :-2])    # 切除结尾 2 个
```

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

当明确隔几个元素采样一次时，示例如下。

In [4]:
```
print(arr1)
print(arr1[  :  :2])     # 每 2 个元素采样一次
print(arr1[  :  :3])     # 每 3 个元素采样一次
print(arr1[1:-1:2])      # 切除一头一尾后，每 2 个元素采样一次
```

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

（2）矩阵的切片

```
In [1] : import numpy as np
```

```
In [2] : arr2 = np.arange(1,21).reshape(4,5)
         print( arr2 )
```

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

```
In [3] : print( arr2[ 1:3 , 1:-1 ] )      # 矩阵切片初体验
```

```
[[  7  8  9]
 [12 13 14]]
```

```
In [4] : print( arr2[ ::3 , ::2 ] )      # 跳跃采样
```

```
[[  1  3  5]
 [16 18 20]]
```

（3）提取矩阵的行

基于矩阵的切片功能，我们可以提取其部分行，如示例所示。

```
In [1] : import numpy as np
```

```
In [2] : arr3 = np.arange(1,21).reshape(4,5)
         print( arr3 )
```

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

```
In [3] : print( arr3[ 2 , : ] )       # 提取第 2 行
```

```
[11 12 13 14 15]
```

```
In [4] : print( arr3[ 1:3 , : ] )      # 提取 1 至 2 行
```

```
[[ 6  7  8  9 10]
 [11 12 13 14 15]]
```

考虑代码的简洁，当提取矩阵的某几行时可简写（但提取列的时候不可简写）。

```
In [5] : print( arr3[ 2 , : ] )      # 规范的提取行
         print( arr3[2] )          # 简便的提取行
```

```
[11 12 13 14 15]
[11 12 13 14 15]
```

所以，有时你可能看到诸如 arr[1][2] 这样的语法，不必吃惊，其实这只是先提取了第 1 行，再提取该行中第 2 个元素。提一句，UP 并不推荐这样的写法。

（4）提取矩阵的列

基于矩阵的切片功能，我们可以提取其部分列，如示例所示。

In [1] :
```
import numpy as np
```

In [2] :
```
arr4 = np.arange(1,21).reshape(4,5)
print( arr4 )
```
```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

In [3] :
```
print( arr4[ : , 2 ] )          # 提取第 2 列（注意，输出的是向量）
```
```
[3 8 13 18]
```

In [4] :
```
print( arr4[ : , 1:3 ] )        # 提取 1 至 2 列
```
```
[[ 2  3]
 [ 7  8]
 [12 13]
 [17 18]]
```

值得注意的是，提取某一个单独的列时，出来的结果是一个向量。其实这么做只是为了省空间，我们知道，列矩阵必须用两层中括号来存储，而形状为 1000 的向量，自然比形状为(1000,1)的列矩阵更省空间（节约了 1000 对括号）。

如果你真的想要提取一个列矩阵出来，示例如下。

In [1] :
```
import numpy as np
```

In [2] :
```
arr5 = np.arange(1,16).reshape(3,5)
print( arr5 )
```
```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
```

In [3] :
```
cut = arr5[ : , 2 ]             # 提取第 2 列为向量
print( cut )
```
```
[3 8 13]
```

In [4] :
```
cut = cut.reshape( (-1,1) )     # 升级为列矩阵
print(cut)
```
```
[[ 3]
 [ 8]
 [13]]
```

3.4 数组切片仅是视图

（1）数组切片仅是视图

与 Python 列表和 Matlab 不同，NumPy 数组的切片仅仅是原数组的一个视图。换言之，NumPy 切片并不会创建新的变量，示例如下。

In [1]:
```
import numpy as np
```

In [2]:
```
arr = np.arange(10)      # 创建原数组 arr
print(arr)
```
```
[0 1 2 3 4 5 6 7 8 9]
```

In [3]:
```
cut = arr[  :3 ]         # 创建 arr 的切片 cut
print(cut)
```
```
[0 1 2]
```

In [4]:
```
cut[0] = 100             # 对切片的数值进行修改
print(cut)
```
```
[100   1   2]
```

In [5]:
```
print(arr)               # 原数组也被修改
```
```
[100   1   2   3   4   5   6   7   8   9]
```

习惯 Matlab 的用户可能无法理解，但其实这正是 NumPy 的精妙之处。试想一下，一个几百万条数据的数组，每次切片时都创建一个新变量，势必造成大量的内存浪费。因此，NumPy 的切片被设计为原数组的视图是极好的。

深度学习中为节省内存，将多次使用 arr[:]=<表达式> 来替代 arr=<表达式>。

（2）备份切片为新变量

如果真的需要为切片创建新变量（这种情况很稀少），使用 .copy() 方法。

In [1]:
```
import numpy as np
```

In [2]:
```
arr = np.arange(10)          # 创建一个 0 到 10 的向量 arr
print(arr)
```
```
[0 1 2 3 4 5 6 7 8 9]
```

In [3]:
```
copy = arr[  :3 ].copy()     # 创建 arr 的拷贝切片
print(copy)
```
```
[0 1 2]
```

In [4]:
```
copy [0] = 100               # 对拷贝切片的数值进行修改
print(copy)
```
```
[100   1   2]
```

In [5]:
```
print(arr)                   # 原数组不为所动
```
```
[0 1 2 3 4 5 6 7 8 9]
```

3.5 数组赋值仅是绑定

（1）数组赋值仅是绑定

与 NumPy 数组的切片一样，NumPy 数组完整的赋值给另一个数组，也只是绑定。换言之，NumPy 数组之间的赋值并不会创建新的变量，示例如下。

```
In [1]: import numpy as np

In [2]: arr1 = np.arange(10)          # 创建一个 0 到 10 的数组变量 arr
        print(arr1)
[0 1 2 3 4 5 6 7 8 9]

In [3]: arr2 = arr1                   # 把数组 1 赋值给另一个数组 2
        print(arr2)
[0 1 2 3 4 5 6 7 8 9]

In [4]: arr2[0] = 100                 # 修改数组 2
        print(arr2)
[100   1   2   3   4   5   6   7   8   9]

In [5]: print(arr1)                   # 原数组也被修改
[100   1   2   3   4   5   6   7   8   9]
```

此特性的出现仍然是为了节约空间，破局的方法仍然与前面相同。

（2）复制数组为新变量

如果真的需要赋给一个新数组，使用 .copy() 方法。

```
In [1]: import numpy as np

In [2]: arr1 = np.arange(10)          # 创建一个 0 到 10 的数组变量 arr
        print(arr1)
[0 1 2 3 4 5 6 7 8 9]

In [3]: arr2 = arr1.copy()            # 把数组 1 的拷贝赋值给另一个数组 2
        print(arr2)
[0 1 2 3 4 5 6 7 8 9]

In [4]: arr2[0] = 100                 # 修改数组 2
        print(arr2)
[100   1   2   3   4   5   6   7   8   9]

In [5]: print(arr1)                   # 查看数组 1
[0 1 2 3 4 5 6 7 8 9]
```

四、数组的变形

4.1 数组的转置

数组的转置方法为 .T，其只对矩阵有效，因此遇到向量要先将其转化为矩阵。

（1）向量的转置

In [1] :
```
import numpy as np
```

In [2] :
```
arr1 = np.arange(1,4)          # 创建向量
print(arr1)
```
```
[1 2 3]
```

In [3] :
```
arr2 = arr1.reshape( (1,-1) )      # 升级为矩阵
print(arr2)
```
```
[[1 2 3]]
```

In [4] :
```
arr3 = arr2.T                  # 行矩阵的转置
print(arr3)
```
```
[[1]
 [2]
 [3]]
```

（2）矩阵的转置

行矩阵的转置刚演示了，列矩阵的转置如示例所示。

In [1] :
```
import numpy as np
```

In [2] :
```
arr1 = np.arange(3).reshape(3,1)      # 创建列矩阵
print(arr1)
```
```
[[0]
 [1]
 [2]]
```

In [3] :
```
arr2 = arr1.T             # 列矩阵的转置
print(arr2)               # 结果为行矩阵
```
```
[[0 1 2]]
```

矩阵的转置如示例所示。

In [1] :
```
import numpy as np
```

In [2] :
```
arr1 = np.arange(4).reshape(2,2)      # 创建矩阵
print(arr1)
```
```
[[0 1]
 [2 3]]
```

In [3] :
```
arr2 = arr1.T             # 矩阵的转置
print(arr2)
```
```
[[0 2]
 [1 3]]
```

4.2 数组的翻转

数组的翻转方法有两个，一个是上下翻转的 np.flipud( ) ，表示 up-down；一个是左右翻转的 np.fliplr( )，表示 left-right。其中，向量只能使用 np.flipud()，在数学中，向量并不是横着排的，而是竖着排的。

（1）向量的翻转

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建向量
arr1 = np.arange(10)
print( arr1 )
```
```
[0 1 2 3 4 5 6 7 8 9]
```

In [3] :
```
# 翻转向量
arr_ud = np.flipud(arr1)
print( arr_ud )
```
```
[9 8 7 6 5 4 3 2 1 0]
```

（2）矩阵的翻转

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建矩阵
arr2 = np.arange(1,21).reshape(4,5)
print( arr2 )
```
```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

In [3] :
```
# 左右翻转
arr_lr = np.fliplr(arr2)
print( arr_lr )
```
```
[[ 5  4  3  2  1]
 [10  9  8  7  6]
 [15 14 13 12 11]
 [20 19 18 17 16]]
```

In [4] :
```
# 上下翻转
arr_ud = np.flipud(arr2)
print( arr_ud )
```
```
[[16 17 18 19 20]
 [11 12 13 14 15]
 [ 6  7  8  9 10]
 [ 1  2  3  4  5]]
```

4.3 数组的重塑

想要重塑数组的形状，需要用到 .reshape() 方法。

前面说过，给定了其他维度的数值，剩下一个维度可以填-1，让它自己去计算。比如把一个 5 行 6 列的矩阵重塑为 3 行 10 列的矩阵，当列的参数 10 告诉它，行的参数直接可以用-1 来替代，它会自己去用 30 除以 10 来计算。

（1）向量的变形

In [1] :
```
import numpy as np
```

In [2] :
```
arr1 = np.arange(1,10)        # 创建向量
print(arr1)
```
```
[1 2 3 4 5 6 7 8 9]
```

In [3] :
```
arr2 = arr1.reshape(3,3)      # 变形为矩阵
print(arr2)
```
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

（2）矩阵的变形

In [1] :
```
import numpy as np
```

In [2] :
```
arr1 = np.array( [ [1,2,3],[4,5,6] ] )      # 创建矩阵
print(arr1)
```
```
[[1 2 3]
 [4 5 6]]
```

In [3] :
```
arr2 = arr1.reshape(6)              # 变形为向量
print(arr2)
```
```
[1 2 3 4 5 6]
```

In [4] :
```
arr3 = arr1.reshape(1,6)            # 变形为矩阵
print(arr3)
```
```
[[1 2 3 4 5 6]]
```

4.4 数组的拼接

（1）向量的拼接

两个向量拼接，将得到一个新的加长版向量。

In [1]:
```
import numpy as np
```

In [2]:
```
# 创建向量 1
arr1 = np.array( [1,2,3] )
print(arr1)
```
```
[1 2 3]
```

In [3]:
```
# 创建向量 2
arr2 = np.array( [4,5,6] )
print(arr2)
```
```
[4 5 6]
```

In [4]:
```
# 拼接
arr3 = np.concatenate( [arr1,arr2] )
print(arr3)
```
```
[1 2 3 4 5 6]
```

（2）矩阵的拼接

两个矩阵可以按不同的维度进行拼接，但拼接时必须注意维度的吻合。

In [1]:
```
import numpy as np
```

In [2]:
```
# 创建数组 1
arr1 = np.array( [[1,2,3],[ 4,5,6]] )
print(arr1)
```
```
[[1 2 3]
 [4 5 6]]
```

In [3]:
```
# 创建数组 2
arr2 = np.array( [[7,8,9],[10,11,12]] )
print(arr2)
```
```
[[ 7  8  9]
 [10 11 12]]
```

In [4]:
```
# 按第一个维度（行）拼接
arr3 = np.concatenate( [arr1,arr2] )          # 默认参数 axis=0
print(arr3)
```
```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

In [5]:
```
# 按第二个维度（列）拼接
arr4 = np.concatenate( [arr1,arr2] ,axis=1 )
print(arr4)
```
```
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

最后要说明的是，向量和矩阵不能进行拼接，必须先把向量升级为矩阵。

4.5 数组的分裂

（1）向量的分裂

向量分裂，将得到若干个更短的向量。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建向量
arr = np.arange(10,100,10)
print(arr)
```
```
[10 20 30 40 50 60 70 80 90]
```

In [3] :
```
# 分裂数组
arr1,arr2,arr3 = np.split( arr , [2,8] )
print(arr1)
print(arr2)
print(arr3)
```
```
[10 20]
[30 40 50 60 70 80]
[90]
```

np.split()函数中，给出的第二个参数[2,8]表示在索引[2]和索引[8]的位置截断。

（2）矩阵的分裂

矩阵的分裂同样可以按不同的维度进行，分裂出来的均为矩阵。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建矩阵
arr = np.arange(1,9).reshape(2,4)
print(arr)
```
```
[[1 2 3 4]
 [5 6 7 8]]
```

In [3] :
```
# 按第一个维度（行）分裂
arr1,arr2 = np.split(arr,[1])                    # 默认参数 axis=0
print(arr1 , '\n\n' , arr2)                      # 注意输出的是矩阵
```
```
[[1 2 3 4]]

[[5 6 7 8]]
```

In [4] :
```
# 按第二个维度（列）分裂
arr1,arr2,arr3 = np.split( arr , [1,3] , axis=1 )
print( arr1 , '\n\n' , arr2 , '\n\n' , arr3 )
```
```
[[1]
 [5]]

[[2 3]
 [6 7]]

[[4]
 [8]]
```

五、数组的运算

5.1 数组与系数之间的运算

Python 基础中，常用的运算符如表 5-1 所示，NumPy 的运算符与之相同。

表 5-1 常见的运算符

运算符	含义	输入	输出
+、-、*、/	加、减、乘、除	1 * 2 + 3 / 4	2.75
**	幂	2 ** 4	16
()	修正运算次序	1 * (2 + 3) / 4	1.25
//	取整	28 // 5	5
%	取余	28 % 5	3

这里仅以矩阵为例，向量与系数的操作与之相同。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建矩阵
arr = np.arange(1,9).reshape(2,4)
print( arr )
```
```
[[1 2 3 4]
 [5 6 7 8]]
```

In [3] :
```
print( arr + 10 )          # 加法
```
```
[[11 12 13 14]
 [15 16 17 18]]
```

In [4] :
```
print( arr - 10 )          # 减法
```
```
[[-9 -8 -7 -6]
 [-5 -4 -3 -2]]
```

In [5] :
```
print( arr * 10 )          # 乘法
```
```
[[10 20 30 40]
 [50 60 70 80]]
```

In [6] :
```
print( arr / 10 )          # 除法
```
```
[[0.1 0.2 0.3 0.4]
 [0.5 0.6 0.7 0.8]]
```

In [7] :
```
print( arr ** 2)           # 平方
```
```
[[ 1  4  9 16]
 [25 36 49 64]]
```

In [8] :
```
print( arr // 6)           # 取整
```
```
[[0 0 0 0]
 [0 1 1 1]]
```

In [9] :
```
print( arr % 6)            # 取余
```
```
[[1 2 3 4]
 [5 0 1 2]]
```

5.2 数组与数组之间的运算

同维度数组间的运算即对应元素之间的运算，这里仅以矩阵为例，向量与向量的操作与之相同。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建矩阵
arr1 = np.arange(-1,-9,-1).reshape(2,4)
arr2 = -arr1
print( arr1 )
print( arr2 )
```
```
[[-1 -2 -3 -4]
 [-5 -6 -7 -8]]
[[1 2 3 4]
 [5 6 7 8]]
```

In [3] :
```
print( arr1 + arr2 )          # 加法
```
```
[[0 0 0 0]
 [0 0 0 0]]
```

In [4] :
```
print( arr1 - arr2 )          # 减法
```
```
[[ -2  -4  -6  -8]
 [-10 -12 -14 -16]]
```

In [5] :
```
print( arr1 * arr2 )          # 乘法
```
```
[[ -1  -4  -9 -16]
 [-25 -36 -49 -64]]
```

In [6] :
```
print( arr1 / arr2 )          # 除法
```
```
[[-1. -1. -1. -1.]
 [-1. -1. -1. -1.]]
```

In [7] :
```
print( arr1 ** arr2)          # 幂方
```
```
[[     -1       4     -27     256]
 [ -3125   46656 -823543 16777216]]
```

上述 In [5]中，乘法是遵循对应元素相乘的，你可以称之为“逐元素乘积”。那么如何实现线性代数中的“矩阵级乘法”呢？6.1 会介绍到相关函数。

5.3 广播

5.2 是同形状数组之间的逐元素运算，本节讲解不同形状的数组之间的运算。本课件仅讨论二维数组之内的情况，不同形状的数组之间的运算有以下规则：

- 如果是向量与矩阵之间做运算，向量自动升级为行矩阵；
- 如果某矩阵是行矩阵或列矩阵，则其被广播，以适配另一个矩阵的形状。

（1）向量被广播

当一个形状为(x,y)的矩阵与一个向量做运算时，要求该向量的形状必须为 y，运算时向量会自动升级成形状为(1,y)的行矩阵，该形状为(1,y)的行矩阵再自动被广播为形状为(x,y)的矩阵，这样就与另一个矩阵的形状适配了。

In [1]:
```
import numpy as np
```

In [2]:
```
# 向量
arr1 = np.array([-100,0,100])
print(arr1)
```
```
[-100    0    100]
```

In [3]:
```
# 矩阵
arr2 = np.random.random( (10,3) )
print(arr2)
```
```
[[0.60755301  0.47875215  0.70909527]
 [0.12946037  0.78380689  0.7771824 ]
 [0.2658308   0.34287368  0.21176781]
 [0.93920876  0.73860266  0.32531675]
 [0.92474339  0.97997977  0.98410076]
 [0.52791609  0.17325381  0.04736612]
 [0.8543378   0.707902    0.36518268]
 [0.72053659  0.71830332  0.12972364]
 [0.86540524  0.95537187  0.66062319]
 [0.46449401  0.88824093  0.77800761]]
```

In [4]:
```
# 广播
print(arr1*arr2)
```
```
[[-60.75530134   0.         70.90952713]
 [-12.94603684   0.         77.71823953]
 [-26.58307957   0.         21.17678114]
 [-93.92087564   0.         32.53167491]
 [-92.47433933   0.         98.41007632]
 [-52.7916095    0.          4.73661185]
 [-85.43377956   0.         36.51826765]
 [-72.05365932   0.         12.97236352]
 [-86.54052368   0.         66.06231879]
 [-46.44940114   0.         77.80076055]]
```

（2）列矩阵被广播

当一个形状为(x,y)的矩阵与一个列矩阵做运算时，要求该列矩阵的形状必须为(x,1)，该形状为(x,1)的列矩阵再自动被广播为形状为(x,y)的矩阵，这样就与另一个矩阵的形状适配了。

In [1]:
```
import numpy as np
```

In [2]:
```
# 列矩阵
arr1 = np.arange(3).reshape(3,1)
print(arr1)
```
```
[[0]
 [1]
 [2]]
```

In [3]:
```
# 矩阵
arr2 = np.ones( (3,5) )
print(arr2)
```
```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

In [4]:
```
# 广播
print(arr1*arr2)
```
```
[[0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2.]]
```

（3）行矩阵与列矩阵同时被广播

当一个形状为(1,y)的行矩阵与一个形状为(x,1) 的列矩阵做运算时，这俩矩阵都会被自动广播为形状为(x,y)的矩阵，这样就互相适配了。

In [1]:
```
import numpy as np
```

In [2]:
```
# 向量
arr1 = np.arange(3)
print(arr1)
```
```
[0 1 2]
```

In [3]:
```
# 列矩阵
arr2 = np.arange(3).reshape(3,1)
print(arr2)
```
```
[[0]
 [1]
 [2]]
```

In [4]:
```
# 广播
print(arr1*arr2)
```
```
[[0 0 0]
 [0 1 2]
 [0 2 4]]
```

六、数组的函数

6.1 矩阵乘积

- 第五章中的乘法都是“逐元素相乘”，这里介绍线性代数中的矩阵乘积，本节只需要使用 np.dot( ) 函数。
- 当矩阵乘积中混有向量时，根据需求，其可充当行矩阵，也可充当列矩阵，但混有向量时输出结果必为向量。

（1）向量与向量的乘积

设两个向量的形状按前后顺序分别是 5 以及 5 。从矩阵乘法的角度，有 $(1,5)\times(5,1)=(1,1)$，因此输出的应该是形状为 1 的向量。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建向量
arr1 = np.arange(5)
arr2 = np.arange(5)
print(arr1)
print(arr2)
```
```
[0 1 2 3 4]
[0 1 2 3 4]
```

In [3] :
```
# 矩阵乘积
print( np.dot(arr1,arr2) )
```
```
30
```

（2）向量与矩阵的乘积

设向量的形状是 5，矩阵的形状是 (5,3) 。从矩阵乘法的角度，有 $(1,5)\times(5,3)=(1,3)$，因此输出的应该是形状为 3 的向量。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建数组
arr1 = np.arange(5)
arr2 = np.arange(15).reshape(5,3)
print(arr1)
print(arr2)
```
```
[0 1 2 3 4]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

In [3] :
```
# 矩阵乘积
print( np.dot(arr1,arr2) )
```
```
[90 100 110]
```

（3）矩阵与向量的乘积

设矩阵的形状是 (3,5)，向量的形状是 5。从矩阵乘法的角度，有 $(3,5)\times(5,1)=(3,1)$，因此输出的应该是形状为 3 的向量。

In [1]:
```
import numpy as np
```

In [2]:
```
# 创建数组
arr1 = np.arange(15).reshape(3,5)
arr2 = np.arange(5)
print(arr1)
print(arr2)
```

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

In [3]:
```
# 矩阵乘积
print( np.dot(arr1,arr2) )
```

```
[ 30  80 130]
```

（4）矩阵与矩阵的乘积

设两个矩阵的形状按前后顺序分别是 (5,2) 以及 (2,8)。从矩阵乘法的角度，有 $(5,2)\times(2,8)=(5,8)$，因此输出的应该是形状为 (5,8) 的矩阵。

In [1]:
```
import numpy as np
```

In [2]:
```
# 创建数组 1
arr1 = np.arange(10).reshape(5,2)
print(arr1)
```

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

In [3]:
```
# 创建数组 2
arr2 = np.arange(16).reshape(2,8)
print(arr2)
```

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

In [4]:
```
# 矩阵乘积
print( np.dot(arr1,arr2) )
```

```
[[  8   9  10  11  12  13  14  15]
 [ 24  29  34  39  44  49  54  59]
 [ 40  49  58  67  76  85  94 103]
 [ 56  69  82  95 108 121 134 147]
 [ 72  89 106 123 140 157 174 191]]
```

6.2 数学函数

NumPy 设计了很多数学函数，这里列举其中最重要、最常见的几个。

In [1] :
```
import numpy as np
```

In [2] :
```
# 绝对值函数
arr_v = np.array( [-10, 0, 10] )
abs_v = np.abs(arr_v)
print( '原数组是：', arr_v )
print( '绝对值是：', abs_v )
```
```
原数组是： [-10   0  10]
绝对值是： [10  0  10]
```

In [3] :
```
# 三角函数
theta = np.arange(3) * np.pi / 2
sin_v = np.sin(theta)
cos_v = np.cos(theta)
tan_v = np.tan(theta)
print( '原数组是：', theta )
print( '正弦值是：', sin_v )
print( '余弦值是：', cos_v )
print( '正切值是：', tan_v )
```
```
原数组是： [0.          1.57079633 3.14159265]
正弦值是： [0.0000000e+00 1.0000000e+00 1.2246468e-16]
余弦值是： [ 1.000000e+00  6.123234e-17 -1.000000e+00]
正切值是： [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

In [4] :
```
# 指数函数
x = np.arange(1,4)
print( 'x     =', x )
print( 'e^x   =', np.exp(x) )
print( '2^x   =', 2**x )
print( '10^x = ', 10**x )
```
```
x     = [1 2 3]
e^x   = [ 2.71828183  7.3890561  20.08553692]
2^x   = [2 4 8]
10^x = [ 10  100 1000]
```

In [5] :
```
# 对数函数
x = np.array( [1,10,100,1000] )
print( 'x          =', x )
print( 'ln(x)      =', np.log(x) )
print( 'log2(x)    =', np.log(x) / np.log(2) )
print( 'log10(x) =', np.log(x) / np.log(10) )
```
```
x          = [   1   10  100 1000]
ln(x)      = [0.         2.30258509 4.60517019 6.90775528]
log2(x)    = [0.         3.32192809 6.64385619 9.96578428]
log10(x) = [0. 1. 2. 3.]
```

6.3 聚合函数

聚合很有用，这里用矩阵演示。向量与之一致，但没有 axis 参数。以下在注释中介绍了 6 个最重要的聚合函数，其用法完全一致，仅演示其中 3 个。

In [1]:
```
import numpy as np
```

In [2]:
```
# 最大值函数 np.max( )与最小值函数 np.min( )
arr = np.random.random((2,3))
print( arr )
print( '按维度一求最大值：', np.max(arr,axis=0) )
print( '按维度二求最大值：', np.max(arr,axis=1) )
print( '整体求最大值：', np.max(arr) )
```

```
[[0.54312818  0.57067295  0.11898755]
 [0.85857494  0.33915753  0.4742594 ]]
按维度一求最大值： [0.85857494  0.57067295  0.4742594 ]
按维度二求最大值： [0.57067295  0.85857494]
整体求最大值： 0.8585749445359108
```

In [3]:
```
# 求和函数 np.sum( )与求积函数 np.prod( )
arr = np.arange(10).reshape(2,5)
print( arr )
print( '按维度一求和：', np.sum(arr,axis=0) )
print( '按维度二求和：', np.sum(arr,axis=1) )
print( '整体求和：', np.sum(arr) )
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
按维度一求和： [ 5  7  9 11 13]
按维度二求和： [10 35]
整体求和： 45
```

In [4]:
```
# 均值函数 np.mean( )与标准差函数 np.std( )
arr = np.arange(10).reshape(2,5)
print( arr )
print( '按维度一求平均：', np.mean(arr,axis=0) )
print( '按维度二求平均：', np.mean(arr,axis=1) )
print( '整体求平均：', np.mean(arr) )
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
按维度一求平均： [2.5 3.5 4.5 5.5 6.5]
按维度二求平均： [2. 7.]
整体求平均： 4.5
```

- 当 axis=0 时，最终结果与每一行的元素个数一致；
  当 axis=1 时，最终结果与每一列的元素个数一致。
- 考虑到大型数组难免有缺失值，以上聚合函数碰到缺失值时会报错，因此出现了聚合函数的安全版本，即计算时忽略缺失值：np.nansum( )、np.nanprod( )、np.nanmean()、np.nanstd()、np.nanmax()、np.nanmin()。

七、布尔型数组

除了整数型数组和浮点型数组，还有一种有用的数组类型——布尔型数组。

7.1 创建布尔型数组

由于 NumPy 的主要数据类型是整数型数组或浮点型数组，因此布尔型数组的产生离不开：大于>、大于等于>=、等于==、不等号!=、小于<、小于等于<=。

首先，我们将数组与系数作比较，以产生布尔型数组，示例如下。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建数组
arr = np.arange(1,7).reshape(2,3)
print(arr)
```
```
[[1 2 3]
 [4 5 6]]
```

In [3] :
```
# 数组与数字作比较
print( arr >= 4 )
```
```
[[False  False  False]
 [True  True  True]]
```

其次，我们将同维数组作比较，以产生布尔型数组，示例如下。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建同维数组
arr1 = np.arange(1,6)
arr2 = np.flipud(arr1)
print(arr1)
print(arr2)
```
```
[1 2 3 4 5]
[5 4 3 2 1]
```

In [3] :
```
# 同维度数组作比较
print( arr1 > arr2)
```
```
[False  False  False  True  True]
```

最后，还可以同时比较多个条件。Python 基础里，同时检查多个条件使用的与、或、非是 **and**、**or**、**not**。但 NumPy 中使用的与、或、非是 **&** 、 **|** 、 **~** 。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建数组
arr = np.arange(1,10)
print(arr)
```
```
[1 2 3 4 5 6 7 8 9]
```

In [3] :
```
# 多个条件
print( (arr < 4) | (arr > 6) )
```
```
[ True  True  True  False  False  False  True  True  True]
```

7.2 布尔型数组中 True 的数量

有三个关于 **Ture** 数量的有用函数，分别是 np.sum()、np.any()、np.all()。

np.sum()函数：统计布尔型数组里 **True** 的个数。示例如下。

In [1]：
```
import numpy as np
```

In [2]：
```
# 创建一个形状为 10000 的标准正态分布数组
arr = np.random.normal( 0,1,10000 )
```

In [3]：
```
# 统计该分布中绝对值小于 1 的元素个数
num = np.sum( np.abs(arr) < 1 )
print( num )
```
6814

在 In[3]里，np.abs(arr)<1 可以替换为 (arr>-1)&(arr<1) 。此外，最终统计的数量为 6814，其概率近似为 0.6827，这符合统计学中的 3σ 准则。

np.any()函数：只要布尔型数组里含有一个及其以上的 **True**，就返回 **True**。

In [1]：
```
import numpy as np
```

In [2]：
```
# 创建同维数组
arr1 = np.arange(1,10)
arr2 = np.flipud(arr1)
print(arr1)
print(arr2)
```
[1 2 3 4 5 6 7 8 9]
[9 8 7 6 5 4 3 2 1]

In [3]：
```
# 统计这两个数组里是否有共同元素
print( np.any( arr1 == arr2 )  )
```
True

从结果来看，arr1 与 arr2 里含有共同元素，那就是 5。

np.all()函数：当布尔型数组里全是 **True** 时，才返回 **True**，示例如下。

In [1]：
```
import numpy as np
```

In [2]：
```
# 模拟英语六级的成绩，创建 100000 个样本
arr = np.random.normal( 500,70,100000 )
```

In [3]：
```
# 判断是否所有考生的分数都高于 250
print( np.all( arr > 250 )  )
```
False

从结果来看，尽管 3σ 准则告诉我们有 99.73%的考生成绩高于 290 分（290 通过 $500-3\times70$ 计算得到），但仍然有最终成绩低于 250 分的裸考者。

7.3 布尔型数组作为掩码

若一个普通数组和一个布尔型数组的维度相同，可以将布尔型数组作为普通数组的掩码，这样可以对普通数组中的元素作筛选。给出两个示例。

第一个示例，筛选出数组中大于、等于或小于某个数字的元素。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建数组
arr = np.arange(1,13).reshape(3,4)
print(arr)
```
```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [3] :
```
# 数组与数字作比较
print( arr > 4 )
```
```
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]]
```

In [4] :
```
# 筛选出 arr > 4 的元素
print( arr[ arr > 4 ] )
```
```
[ 5  6  7  8  9 10 11 12]
```

注意，这个矩阵进行掩码操作后，退化为了向量。

第二个示例，筛选出数组逐元素比较的结果。

In [1] :
```
import numpy as np
```

In [2] :
```
# 创建同维数组
arr1 = np.arange(1,10)
arr2 = np.flipud(arr1)
print(arr1)
print(arr2)
```
```
[1 2 3 4 5 6 7 8 9]
[9 8 7 6 5 4 3 2 1]
```

In [3] :
```
# 同维度数组作比较
print( arr1 > arr2)
```
```
[False False False False  True  True  True  True]
```

In [4] :
```
# 筛选出 arr1 > arr2 位置上的元素
print( arr1[ arr1 > arr2 ] )
print( arr2[ arr1 > arr2 ] )
```
```
[6 7 8 9]
[4 3 2 1]
```

7.4 满足条件的元素所在位置

现在我们来思考一种情况：假设一个很长的数组，我想知道满足某个条件的元素们所在的索引位置，此时使用 np.where( )函数。

In [1] :
```
import numpy as np
```

In [2] :
```
# 模拟英语六级成绩的随机数组，取 10000 个样本
arr = np.random.normal( 500,70,1000 )
```

In [3] :
```
# 找出六级成绩超过 650 的元素所在位置
print( np.where( arr > 650 ) )
```
```
(array([127, 129, 317, 342, 484, 490, 634, 658, 677, 755, 763, 819, 820,
        853, 926, 932, 982], dtype=int64),)
```

In [4] :
```
# 找出六级成绩最高分的元素所在位置
print( np.where( arr == np.max(arr) ) )
```
```
(array([342], dtype=int64),)
```

np.where()函数的输出看起来比较怪异，它是输出了一个元组。元组第一个元素是“满足条件的元素所在位置”；第二个元素是数组类型，可忽略掉。

八、从数组到张量

本视频中，numpy 为 1.21.5 版本，torch 为 1.12.0 版本。

PyTorch 不同版本的发行日志：https://pytorch.org/blog/。

8.1 数组与张量

- 本次课属于《Python 深度学习》系列视频，PyTorch 作为当前首屈一指的深度学习库，其将 NumPy 的语法尽数吸收，作为自己处理数组的基本语法，且运算速度从使用 CPU 的数组进步到使用 GPU 的张量。
- NumPy 和 PyTorch 的基础语法几乎一致，具体表现为：

  ① np 对应 torch；

  ② 数组 array 对应张量 tensor；

  ③ NumPy 的 n 维数组对应着 PyTorch 的 n 阶张量。
- 数组与张量之间可以相互转换：

  数组 arr 转为张量 ts：ts = torch.tensor(arr)；

  张量 ts 转为数组 arr：arr = np.array(ts)。

8.2 语法不同点

为了找到 NumPy 和 PyTorch 哪些语法不同，UP 对本文档进行了替换操作，将 np 改为 torch，将 array 改为 tensor，并重新运行所有代码，得出结论：PyTorch 只是少量修改了 NumPy 的部分函数或方法，现对其中不同的地方进行罗列。

表 8-1　PyTorch 修正的 NumPy 函数或方法

课件位置	NumPy 的函数	PyTorch 的函数	用法区别
1.1 数据类型	.astype( )	.type( )	无
2.4 随机数组	np.random.random( )	torch.rand( )	无
2.4 随机数组	np.random.randint( )	torch.randint( )	不接纳一维张量
2.4 随机数组	np.random.normal( )	torch.normal( )	不接纳一维张量
2.4 随机数组	np.random.randn( )	torch.randn( )	无
3.4 数组切片	.copy( )	.clone( )	无
4.4 数组拼接	np.concatenate( )	torch.cat( )	无
4.5 数组分裂	np.split( )	torch.split( )	参数含义优化
6.1 矩阵乘积	np.dot( )	torch.matmul( )	无
6.1 矩阵乘积	np.dot(v,v)	torch.dot( )	无
6.1 矩阵乘积	np.dot(m,v)	torch.mv( )	无
6.1 矩阵乘积	np.dot(m,m)	torch.mm( )	无
6.2 数学函数	np.exp( )	torch.exp( )	必须传入张量
6.2 数学函数	np.log( )	torch.log( )	必须传入张量
6.3 聚合函数	np.mean( )	torch.mean( )	必须传入浮点型张量
6.3 聚合函数	np.std( )	torch.std( )	必须传入浮点型张量