< 理解Python中的数据类型 | 目录 | 使用Numpy计算：通用函数 >

Open in Colab

The Basics of NumPy Arrays

NumPy数组基础

> Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter 3) are built around the NumPy array. This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. Get to know them well!

Python中的数据操作基本就是NumPy数组操作的同义词：一些新的工具像Pandas（第三章）都是依赖于NumPy数组建立起来的。本节会展示使用NumPy数组操作和访问数据以及子数组的一些例子，包括切分、变形和组合。尽管这里展示的操作有些枯燥和学术化，但是它们是组成本书后面使用的例子的基础。你应该更好的掌握它们。

> We'll cover a few categories of basic array manipulations here:
>
> - *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
> - *Indexing of arrays*: Getting and setting the value of individual array elements
> - *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
> - *Reshaping of arrays*: Changing the shape of a given array
> - *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

我们会讨论下述数组操作的基本内容：

- *数组的属性*: 获得数组的大小、形状、内存占用以及数据类型
- *数组索引*: 获得和设置单个数组元素的值
- *数组切片*: 获得和设置数组中的子数组
- *数组变形*: 改变数组的形状
- *组合和切分数组*: 将多个数组组合成一个，或者将一个数组切分成多个

NumPy Array Attributes

NumPy数组属性

> First let's discuss some useful array attributes. We'll start by defining three random arrays, a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

首先我们来讨论一些数组有用的属性。我们从定义三个数组开始，一个一维的，一个二维的和一个三维的数组。我们采用NumPy的随机数产生器来创建数组，产生之前我们会给定一个随机种子，这样来保证每次代码运行的时候都能得到相同的数组：

In [1]:
```python
import numpy as np
np.random.seed(0)  # 设定随机种子，保证实验的可重现

x1 = np.random.randint(10, size=6)  # 一维数组
x2 = np.random.randint(10, size=(3, 4))  # 二维数组
x3 = np.random.randint(10, size=(3, 4, 5))  # 三维数组
```

> Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

每个数组都有属性`ndim`，代表数组的维度，`shape`代表每个维度的长度（形状）和`size`代表数组的总长度（元素个数）

In [2]:
```python
# 输出三维数组的维度、形状和总长度
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

> Another useful attribute is the `dtype`, the data type of the array (which we discussed previously in Understanding Data Types in Python):

另一个有用的属性是`dtype`，数组的数据类型（我们在上一节理解Python的数据类型中已经见过）。

In [3]:
```python
print("dtype:", x3.dtype)
```

```
dtype: int64
```

> Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

还有属性包括`itemsize`代表每个数组元素的长度（单位字节），`nbytes`代表数组的总字节长度：

In [4]:
```python
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 8 bytes
nbytes: 480 bytes
```

> In general, we expect that `nbytes` is equal to `itemsize` times `size`.

通常，我们可以认为`nbytes`等于`itemsize`乘以`size`。

Array Indexing: Accessing Single Elements

数组索引：获取单个元素

> If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

如果我们熟悉Python列表的索引方式，那么NumPy数组的索引方式也是很相似的。对于一维数组来说，第i个元素值（从0开始）可以使用中括号内的索引值获得：

In [5]:
```python
x1
```

Out[5]:
```
array([5, 0, 3, 3, 7, 9])
```

In [6]:
```python
x1[0]
```

Out[6]:
```
5
```

In [7]:
```python
x1[4]
```

Out[7]:
```
7
```

> To index from the end of the array, you can use negative indices:

需要从末尾进行索引取值，你可以使用负的索引值：

In [8]:
```python
x1[-1]
```

Out[8]:
```
9
```

In [9]:
```python
x1[-2]
```

Out[9]:
```
7
```

> In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

在多维数组中获取元素值，可以在中括号中使用一个索引值的元组：

译者注：多维数组的索引方式与列表的列表索引方式是不同的。列表的列表在Python中需要使用多个中括号进行索引，如`x[i][j]`的方式。

In [10]:
```python
x2
```

Out[10]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [11]:
```python
x2[0, 0]
```

Out[11]:
```
3
```

In [12]:
```python
x2[2, 0]
```

Out[12]:
```
1
```

In [13]:
```python
x2[2, -1]
```

Out[13]:
```
7
```

> Values can also be modified using any of the above index notation:

元素值也可以通过上述的索引语法进行修改：

In [14]:
```python
x2[0, 0] = 12
x2
```

Out[14]:
```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

> Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

请记住，与Python的列表不同，NumPy数组是固定类型的。这意味着，如果你试图将一个浮点数值放入一个整数型数组，这个值会被默默地截成整数。这是比较容易犯的错误。

In [15]:
```python
x1[0] = 3.14159  # 会被截成整数
x1
```

Out[15]:
```
array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

数组切片：获取子数组

> Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use this:

```
x[start:stop:step]
```

> If any of these are unspecified, they default to the values `start=0`, `stop=`*`size of dimension`*, `step=1`. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

正如我们可以使用中括号获取单个元素值，我们也可以使用中括号的*切片*语法获取子数组，切片的语法遵从标准Python列表的切片语法格式；对于一个数组`x`进行切片：

```
x[start:stop:step]
```

如果三个参数没有设置值的话，默认值分别是`start=0`，`stop=`*`维度的长度`*，`step=1`。我们来看看在一维数组和多维数组中进行切片取子数组的例子。

One-dimensional subarrays

一维子数组

In [16]:
```python
x = np.arange(10)
x
```

Out[16]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [17]:
```python
x[:5]  # 前五个元素
```

Out[17]:
```
array([0, 1, 2, 3, 4])
```

In [19]:
```python
x[5:]  # 从序号5开始的所有元素
```

Out[19]:
```
array([5, 6, 7, 8, 9])
```

In [20]:
```python
x[4:7]  # 中间4~6序号的元素
```

Out[20]:
```
array([4, 5, 6])
```

In [21]:
```python
x[::2]  # 每隔一个取元素
```

Out[21]:
```
array([0, 2, 4, 6, 8])
```

In [22]:
```python
x[1::2]  # 每隔一个取元素，开始序号为1
```

Out[22]:
```
array([1, 3, 5, 7, 9])
```

> A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

当step为负值时，将会在数组里反向的取元素，这是将数组反向排序最简单的方法：

译者注，从其他编程语言转Python的初学者，很容易问一个问题，我想反序一个字符串，怎么找不到函数啊，内建的没有，str的方法也没有。答案是，因为根本不需要，例如：

```python
s = 'hello world'
# 下面就会输出'dlrow olleh'
print(s[::-1])
```

In [23]:
```python
x[::-1]  # 反序数组
```

Out[23]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

In [24]:
```python
x[5::-2]  # 从序号5开始向前取元素，每隔一个取一个元素
```

Out[24]:
```
array([5, 3, 1])
```

Multi-dimensional subarrays

多维子数组

> Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

多维数组的切片也一样，只是在中括号中使用逗号分隔多个切片声明。例如：

In [25]:
```python
x2
```

Out[25]:
```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [26]:
```python
x2[:2, :3]  # 行的维度取前两个，列的维度取前三个，形状变为(2, 3)
```

Out[26]:
```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

In [27]:
```python
x2[:3, ::2]  # 行的维度取前三个（全部），列的维度每个一个取一列，形状变为(3, 2)
```

Out[27]:
```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

> Finally, subarray dimensions can even be reversed together:

最后，子数组的各维度还可以反序：

In [28]:
```python
x2[::-1, ::-1] # 行和列都反序，形状保持(3, 4)
```

Out[28]:
```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

Accessing array rows and columns

获取数组的行和列

> One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (`:`):

还有一种常见的需要是获取数组的单行或者单列。这可以通过组合索引和切片两个操作做到，使用一个不带参数的冒号`:`可以表示取该维度的所有元素：

In [29]:
```python
print(x2[:, 0])  # x2的第一列
```

```
[12  7  1]
```

In [30]:
```python
print(x2[0, :])  # x2的第一行
```

```
[12  5  2  4]
```

> In the case of row access, the empty slice can be omitted for a more compact syntax:

如果是获取行数据的话，可以省略后续的切片，写成更加简洁的方式：

In [31]:
```python
print(x2[0])  # 等同于 x2[0, :]
```

```
[12  5  2  4]
```

Subarrays as no-copy views

子数组是非副本的视图

> One important–and extremely useful–thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

一个非常重要和有用的概念你需要知道的就是数组的切片返回的实际上是子数组的*视图*而不是它们的副本。这是NumPy数组的切片和Python列表的切片的主要区别，列表的切片返回的是副本。用上面的二维做例子：

In [32]:
```python
print(x2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

> Let's extract a 2×2 subarray from this:

让我们从中取一个2×2的子数组：

In [33]:
```python
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

> Now if we modify this subarray, we'll see that the original array is changed! Observe:

如果我们修改这个子数组，我们看到原来的数组也会随之更改：

In [34]:
```python
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

In [35]:
```python
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

> This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

这个默认行为是很有用的：这意味着当我们在处理大数据集时，我们可以获取和处理其中的部分子数据集而不需要在内存中复制一份数据的副本。

Creating copies of arrays

创建数组的副本

> Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

尽管使用视图有上述的优点，有时候我们还是需要从数组中复制一份子数组出来。这可以使用`copy`方法简单的办到：

In [36]:
```python
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[99  5]
 [ 7  6]]
```

> If we now modify this subarray, the original array is not touched:

现在如果我们改变这个子数组，原数组会保持不变：

In [37]:
```python
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

In [38]:
```python
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

改变数组的形状

> Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

另一个数组的常用操作是改变形状。最方便的方式是使用`reshape`方法实现。例如，如果你希望将1~9的数放入一个3×3的数组里面，你可以这样做：

In [39]:
```python
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

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

> Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with non-contiguous memory buffers this is not always the case.

注意，改变形状要能成功，原始数组和新的形状的数组的总长度`size`必须一样。当可能的情况下，`reshape`会尽量使用原始数组的视图，但是如果原始数组的数据存储在不连续的内存区，就会进行复制。

> Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

另外一个常用的改变形状的操作就是将一个一维数组变成二维数组中的一行或者一列。这也可以使用`reshape`方法实现，或者更简单的方式是使用切片语法中的`newaxis`属性增加一个维度：

In [40]:
```python
x = np.array([1, 2, 3])

# 使用reshape变为 (1, 3)
x.reshape((1, 3))
```

Out[40]:
```
array([[1, 2, 3]])
```

In [41]:
```python
# 使用newaxis，增加行维度，形状也是 (1, 3)
x[np.newaxis, :]
```

Out[41]:
```
array([[1, 2, 3]])
```

In [42]:
```python
# 使用reshape变为 (3, 1)
x.reshape((3, 1))
```

Out[42]:
```
array([[1],
       [2],
       [3]])
```

In [43]:
```python
# 使用newaxis增加列维度，形状也是 (3, 1)
x[:, np.newaxis]
```

Out[43]:
```
array([[1],
       [2],
       [3]])
```

> We will see this type of transformation often throughout the remainder of the book.

我们会在本书后续的内容经常看到这样的变换。

Array Concatenation and Splitting

数组的连接和切分

> All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

前面的方法都是在单个数组上进行操作。我们也可以将多个数组组成一个，或者反过来将一个数组切分成多个。下面我们来看看这些操作。

Concatenation of arrays

连接数组

> Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

在NumPy中连接或者组合多个数组，有三个不同的方法`np.concatenate`，`np.vstack`和`np.hstack`。`np.concatenate`接受一个数组的元组或列表作为第一个参数，如下：

In [44]:
```python
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[44]:
```
array([1, 2, 3, 3, 2, 1])
```

> You can also concatenate more than two arrays at once:

你也可以一次连接两个以上的数组：

In [45]:
```python
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

> It can also be used for two-dimensional arrays:

也可以用来连接二维数组：

In [46]:
```python
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

In [47]:
```python
# 沿着第一个维度进行连接，即按照行连接，axis=0
np.concatenate([grid, grid])
```

Out[47]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In [48]:
```python
# 沿着第二个维度进行连接，即按照列连接，axis=1
np.concatenate([grid, grid], axis=1)
```

Out[48]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

> For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

进行连接的数组如果具有不同的维度，使用`np.vstack`（垂直堆叠）和`np.hstack`（水平堆叠）会更加清晰：

In [49]:
```python
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# 沿着垂直方向进行堆叠
np.vstack([x, grid])
```

Out[49]:
```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

In [50]:
```python
# 沿着水平方向进行堆叠
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Out[50]:
```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

> Similary, `np.dstack` will stack arrays along the third axis.

类似的，`np.dstack`会沿着第三个维度（深度）进行堆叠。

Splitting of arrays

切分数组

> The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

连接的反操作是切分，主要的方法包括`np.split`，`np.hsplit`和`np.vsplit`。我们可以传递一个列表参数表示切分的点：

In [51]:
```python
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5]) # 在序号3和序号5处进行切分，返回三个数组
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

> Notice that *N* split-points, leads to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

你应该已经发现N个切分点会返回N+1个子数组。相应的`np.hsplit`和`np.vsplit`也是相似的：

In [52]:
```python
grid = np.arange(16).reshape((4, 4))
grid
```

Out[52]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [54]:
```python
upper, lower = np.vsplit(grid, [2]) # 沿垂直方向切分，切分点行序号为2
print(upper)
print(lower)
```

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

In [55]:
```python
left, right = np.hsplit(grid, [2]) # 沿水平方向切分数组，切分点列序号为2
print(left)
print(right)
```

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

> Similarly, `np.dsplit` will split arrays along the third axis.

同样`np.dsplit`会沿着第三个维度切分数组。

< 理解Python中的数据类型 | 目录 | 使用Numpy计算：通用函数 >

Open in Colab