02.02-The-Basics-Of-NumPy-Arrays

June 27, 2018

< Understanding Data Types in Python | Contents | Computation on NumPy Arrays: Universal Functions >

1 The Basics of NumPy Arrays

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!

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

1.1 NumPy Array Attributes

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:

```
In [2]: import numpy as np
        np.random.seed(0)  # seed for reproducibility

        x1 = np.random.randint(10, size=6)  # One-dimensional array
        x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
        x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

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

```
In [2]: 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):

```
In [3]: 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:

```
In [4]: 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`.

1.2 Array Indexing: Accessing Single Elements

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:

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

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

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

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

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

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

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

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

Out[8]: 9

In [9]: x1[-2]

Out[9]: 7
```

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

```
In [10]: x2

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

In [11]: x2[0, 0]

Out[11]: 3

In [12]: x2[2, 0]

Out[12]: 1

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

Out[13]: 7
```

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

```
In [14]: 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!

```
In [15]: x1[0] = 3.14159  # this will be truncated!
         x1

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

1.3 Array Slicing: Accessing Subarrays

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.

1.3.1 One-dimensional subarrays

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

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

In [17]: x[:5]  # first five elements

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

In [18]: x[5:]  # elements after index 5

Out[18]: array([5, 6, 7, 8, 9])

In [19]: x[4:7]  # middle sub-array

Out[19]: array([4, 5, 6])

In [20]: x[::2]  # every other element

Out[20]: array([0, 2, 4, 6, 8])

In [21]: x[1::2]  # every other element, starting at index 1

Out[21]: 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:

```
In [22]: x[::-1]  # all elements, reversed

Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In [23]: x[5::-2]  # reversed every other from index 5

Out[23]: array([5, 3, 1])
```

1.3.2 Multi-dimensional subarrays

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

```
In [24]: x2

Out[24]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])

In [25]: x2[:2, :3]  # two rows, three columns
```

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

In [26]: x2[:3, ::2]  # all rows, every other column

Out[26]: array([[12,  2],
                [ 7,  8],
                [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

```
In [27]: x2[::-1, ::-1]

Out[27]: array([[ 7,  7,  6,  1],
                [ 8,  8,  6,  7],
                [ 4,  2,  5, 12]])
```

Accessing array rows and columns 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 (`:`):

```
In [28]: print(x2[:, 0])  # first column of x2

[12  7  1]


In [29]: print(x2[0, :])  # first row of x2

[12  5  2  4]
```

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

```
In [30]: print(x2[0])  # equivalent to x2[0, :]

[12  5  2  4]
```

1.3.3 Subarrays as no-copy views

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:

```
In [31]: print(x2)

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

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

```
In [32]: 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:

```
In [33]: x2_sub[0, 0] = 99
         print(x2_sub)
```

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

```
In [34]: 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.

1.3.4 Creating copies of arrays

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:

```
In [35]: 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:

```
In [36]: x2_sub_copy[0, 0] = 42
         print(x2_sub_copy)
```

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

```
In [37]: print(x2)
```

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

1.4 Reshaping of Arrays

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:

```
In [38]: 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.

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:

```
In [39]: x = np.array([1, 2, 3])

         # row vector via reshape
         x.reshape((1, 3))
```

```
Out[39]: array([[1, 2, 3]])
```

```
In [40]: # row vector via newaxis
         x[np.newaxis, :]
```

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

```
In [41]: # column vector via reshape
         x.reshape((3, 1))
```

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

```
In [42]: # column vector via newaxis
         x[:, np.newaxis]
```

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

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

1.5 Array Concatenation and Splitting

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.

1.5.1 Concatenation of arrays

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:

```
In [43]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])

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

You can also concatenate more than two arrays at once:

```
In [44]: 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:

```
In [45]: grid = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [46]: # concatenate along the first axis
         np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In [47]: # concatenate along the second axis (zero-indexed)
         np.concatenate([grid, grid], axis=1)

Out[47]: 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:

```
In [48]: x = np.array([1, 2, 3])
         grid = np.array([[9, 8, 7],
                          [6, 5, 4]])

         # vertically stack the arrays
         np.vstack([x, grid])

Out[48]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])

In [49]: # horizontally stack the arrays
         y = np.array([[99],
                       [99]])
         np.hstack([grid, y])

Out[49]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

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

1.5.2 Splitting of arrays

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:

```
In [4]: x = [1, 2, 3, 99, 99, 3, 2, 1]
        x1, x2, x3 = np.split(x, [3, 5])
        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:

```
In [7]: grid = np.arange(16).reshape((4, 4))
        grid

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

In [9]: upper, lower = np.vsplit(grid, [2])
        print(upper)
        print(lower)
```

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

```
In [53]: left, right = np.hsplit(grid, [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.

< Understanding Data Types in Python | Contents | Computation on NumPy Arrays: Universal Functions >