02.02-The-Basics-Of-NumPy-Arrays

November 10, 2024

1 The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Part 3) are built around the NumPy array. This chapter 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 values 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 random arrays of one, two, and three dimensions. 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:

[1]:
```
import numpy as np
rng = np.random.default_rng(seed=1701)  # seed for reproducibility

x1 = rng.integers(10, size=6)  # one-dimensional array
x2 = rng.integers(10, size=(3, 4))  # two-dimensional array
x3 = rng.integers(10, size=(3, 4, 5))  # three-dimensional array
```

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

[2]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:   ", x3.dtype)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
dtype:    int64
```

For more discussion of data types, see Understanding Data Types in Python.

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:

[3]:
```
x1
```

[3]:
```
array([9, 4, 0, 3, 8, 6])
```

[4]:
```
x1[0]
```

[4]:
```
9
```

[5]:
```
x1[4]
```

[5]:
```
8
```

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

[6]:
```
x1[-1]
```

[6]:
```
6
```

[7]:
```
x1[-2]
```

[7]:
```
8
```

In a multidimensional array, items can be accessed using a comma-separated `(row, column)` tuple:

[8]:
```
x2
```

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

[9]:
```
x2[0, 0]
```

[9]:
```
3
```

[10]:
```
x2[2, 0]
```

[10]:
```
0
```

[11]:
```
x2[2, -1]
```

[11]:
```
9
```

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

[12]:
```
x2[0, 0] = 12
x2
```

[12]:
```
array([[12,  1,  3,  7],
       [ 4,  0,  2,  3],
       [ 0,  0,  6,  9]])
```

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 into an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

[13]:
```
x1[0] = 3.14159  # this will be truncated!
x1
```

[13]:
```
array([3, 4, 0, 3, 8, 6])
```

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`. Let's look at some examples of accessing subarrays in one dimension and in multiple dimensions.

1.3.1 One-Dimensional Subarrays

Here are some examples of accessing elements in one-dimensional subarrays:

[14]:
```
x1
```

[14]:
```
array([3, 4, 0, 3, 8, 6])
```

[15]:
```
x1[:3]  # first three elements
```

[15]:
```
array([3, 4, 0])
```

[16]:
```
x1[3:]  # elements after index 3
```

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

[17]:
```
x1[1:4]  # middle subarray
```

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

[18]:
```
x1[::2]  # every second element
```

[18]:
```
array([3, 0, 8])
```

[19]:
```
x1[1::2]  # every second element, starting at index 1
```

[19]:
```
array([4, 3, 6])
```

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:

[20]:
```
x1[::-1]  # all elements, reversed
```

[20]:
```
array([6, 8, 3, 0, 4, 3])
```

[21]:
```
x1[4::-2]  # every second element from index 4, reversed
```

[21]:
```
array([8, 0, 3])
```

1.3.2 Multidimensional Subarrays

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

[22]:
```
x2
```

[22]:
```
array([[12,  1,  3,  7],
       [ 4,  0,  2,  3],
       [ 0,  0,  6,  9]])
```

[23]:
```
x2[:2, :3]  # first two rows & three columns
```

[23]:
```
array([[12,  1,  3],
       [ 4,  0,  2]])
```

[24]:
```
x2[:3, ::2]  # three rows, every second column
```

[24]:
```
array([[12,  3],
       [ 4,  2],
       [ 0,  6]])
```

[25]:
```
x2[::-1, ::-1]  # all rows & columns, reversed
```

[25]:
```
array([[ 9,  6,  0,  0],
       [ 3,  2,  0,  4],
       [ 7,  3,  1, 12]])
```

Accessing array rows and columns One commonly needed routine is accessing 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 (:):

[26]:
```
x2[:, 0]  # first column of x2
```

[26]:
```
array([12,  4,  0])
```

[27]:
```
x2[0, :]  # first row of x2
```

[27]:
```
array([12,  1,  3,  7])
```

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

[28]:
```
x2[0]  # equivalent to x2[0, :]
```

[28]:
```
array([12,  1,  3,  7])
```

1.3.3 Subarrays as No-Copy Views

Unlike Python list slices, NumPy array slices are returned as *views* rather than *copies* of the array data. Consider our two-dimensional array from before:

[29]:
```
print(x2)
```

```
[[12  1  3  7]
 [ 4  0  2  3]
 [ 0  0  6  9]]
```

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

[30]:
```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  1]
 [ 4  0]]
```

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

[31]:
```
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  1]
 [ 4  0]]
```

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

```
[[99  1  3  7]
 [ 4  0  2  3]
 [ 0  0  6  9]]
```

Some users may find this surprising, but it can be advantageous: for example, when working 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:

[33]:
```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[99  1]
 [ 4  0]]
```

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

[34]:
```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  1]
 [ 4  0]]
```

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

```
[[99  1  3  7]
 [ 4  0  2  3]
 [ 0  0  6  9]]
```

1.4 Reshaping of Arrays

Another useful type of operation is reshaping of arrays, which can be done 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:

[36]:
```
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, and in most cases the `reshape` method will return a no-copy view of the initial array.

A common reshaping operation is converting a one-dimensional array into a two-dimensional row or column matrix:

[37]:
```
x = np.array([1, 2, 3])
x.reshape((1, 3))  # row vector via reshape
```

[37]:
```
array([[1, 2, 3]])
```

[38]:
```
x.reshape((3, 1))  # column vector via reshape
```

[38]:
```
array([[1],
       [2],
       [3]])
```

A convenient shorthand for this is to use `np.newaxis` in the slicing syntax:

[39]:
```
x[np.newaxis, :]  # row vector via newaxis
```

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

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

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

This is a pattern that we will utilize often throughout the remainder of the book.

1.5 Array Concatenation and Splitting

All of the preceding routines worked on single arrays. NumPy also provides tools to combine multiple arrays into one, and to conversely split a single array into multiple arrays.

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 you can see here:

[41]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

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

You can also concatenate more than two arrays at once:

[42]:
```
z = np.array([99, 99, 99])
print(np.concatenate([x, y, z]))
```

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

And it can be used for two-dimensional arrays:

[43]:
```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

[44]:
```
# concatenate along the first axis
np.concatenate([grid, grid])
```

[44]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

[45]:
```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

[45]:
```
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:

[46]:
```
# vertically stack the arrays
np.vstack([x, grid])
```

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

[47]:
```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

[47]:
```
array([[ 1,  2,  3, 99],
       [ 4,  5,  6, 99]])
```

Similarly, for higher-dimensional arrays, `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:

[48]:
```
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:

[49]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

[49]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
```

```
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[50]:
```
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]]
```

[51]:
```
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, for higher-dimensional arrays, `np.dsplit` will split arrays along the third axis.