The Basics of NumPy Arrays

Data manipulation in Python is nearly **synonymous** with **NumPy array manipulation.**

Tools like **Pandas** (Part 3) are built **around the NumPy array.**

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

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 value in order to ensure that the **same random arrays** are generated **each time this code is run:**

In [4]:
```
import numpy as np
rng = np.random.default_rng(seed=1701)  # seed for reproducibil

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 arra
```

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

Out[7]:
```
array([9, 4, 0, 3, 8, 6], dtype=int64)
```

In [9]:
```
x2
```

Out[9]:
```
array([[3, 1, 3, 7],
       [4, 0, 2, 3],
       [0, 0, 6, 9]], dtype=int64)
```

In [11]:
```
x3
```

Out[11]:
```
array([[[4, 3, 5, 5, 0],
        [8, 3, 5, 2, 2],
        [1, 8, 8, 5, 3],
        [0, 0, 8, 5, 8]],

       [[5, 1, 6, 2, 3],
        [1, 2, 5, 6, 2],
        [5, 2, 7, 9, 3],
        [5, 6, 0, 2, 0]],

       [[2, 9, 4, 3, 9],
        [9, 2, 2, 4, 0],
        [0, 3, 0, 0, 2],
        [3, 2, 7, 4, 7]]], dtype=int64)
```

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):

In [ ]:
```
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
```

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 **ith value** (counting from zero) can be accessed by specifying the **desired index in square brackets,** just as with Python lists:

In [23]:
```
x1
```

```
Out[23]: array([9, 4, 0, 3, 8, 6], dtype=int64)
```

```
In [25]: x1[0]
```

```
Out[25]: 9
```

```
In [27]: x1[4]
```

```
Out[27]: 8
```

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

```
In [30]: x1[-1]
```

```
Out[30]: 6
```

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

```
Out[ ]: 8
```

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

In [ ]:
```
x2
```

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

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

Out[ ]:
```
3
```

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

Out[ ]:
```
0
```

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

Out[ ]:
```
9
```

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

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

Out[ ]:
```
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.**

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

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

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`.

One-Dimensional Subarrays

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

In [ ]:
```
x1
```
Out[ ]:
```
array([3, 4, 0, 3, 8, 6])
```
In [ ]:
```
x1[:3]  # first three elements
```
Out[ ]:
```
array([3, 4, 0])
```
In [ ]:
```
x1[3:]  # elements after index 3
```
Out[ ]:
```
array([3, 8, 6])
```
In [ ]:
```
x1[1:4]  # middle subarray
```
Out[ ]:
```
array([4, 0, 3])
```
In [ ]:
```
x1[::2]  # every second element
```
Out[ ]:
```
array([3, 0, 8])
```

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

Out[ ]:
```
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:**

In [ ]:
```
x1
```

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

In [49]:
```
x1[::-1]  # all elements, reversed
```

Out[49]:
```
array([6, 8, 3, 0, 4, 9], dtype=int64)
```

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

Out[ ]:
```
array([8, 0, 9])
```

In [70]:
```
x1
```

Out[70]:
```
array([9, 4, 0, 3, 8, 6], dtype=int64)
```

In [68]:
```
x1[4:2:-1] # no defults here
```

Out[68]:
```
array([8, 3], dtype=int64)
```

Multidimensional Subarrays

Multidimensional slices work in the **same way,** with multiple slices separated by **commas.**

For example:

In [ ]:
```
x2
```

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

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

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

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

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

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

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

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 (`:`):**

In [ ]:
```
x2
```

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

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

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

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

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

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

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

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

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:

In [74]:
```
print(x2)
```

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

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

In [76]:
```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[3 1]
 [4 0]]
```

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

In [78]:
```
x2_sub[0, 0] = 99
print(x2_sub)
```

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

In [80]:
```
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.

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 [82]:
```
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:**

In [84]:
```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

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

In [86]:
```
print(x2)
```

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

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:

In [ ]:
```
np.arange(1, 10)
```

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

In [ ]:
```
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,**.

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:**

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

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

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

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

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

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

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

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

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

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

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.

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:

In [91]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[91]:
```
array([1, 2, 3, 3, 2, 1])
```

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

In [93]:
```
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:**

In [95]:
```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

In [97]:
```
# concatenate along the first axis
np.concatenate([grid, grid])
```

Out[97]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In [99]:
```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

Out[99]:
```
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 [101…
```
x
```

Out[101… array([1, 2, 3])

In [107…
```
grid
```

Out[107…
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [111…
```
# vertically stack the arrays
np.vstack([x, grid])
```

Out[111…
```
array([[1, 2, 3],
       [1, 2, 3],
       [4, 5, 6]])
```

In [115…
```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Out[115…
```
array([[ 1,  2,  3, 99],
       [ 4,  5,  6, 99]])
```

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 [ ]:
```
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 [121…
```
grid = np.arange(16).reshape((4, 4))
grid
```

Out[121…

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

In [123…

```python
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 [125…

```python
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]]
```