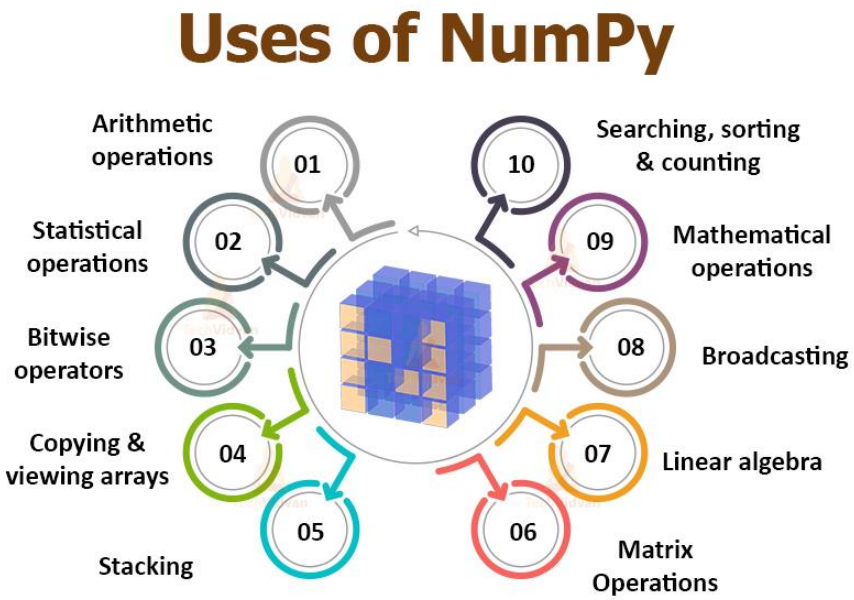
Topic – Numpy arrays



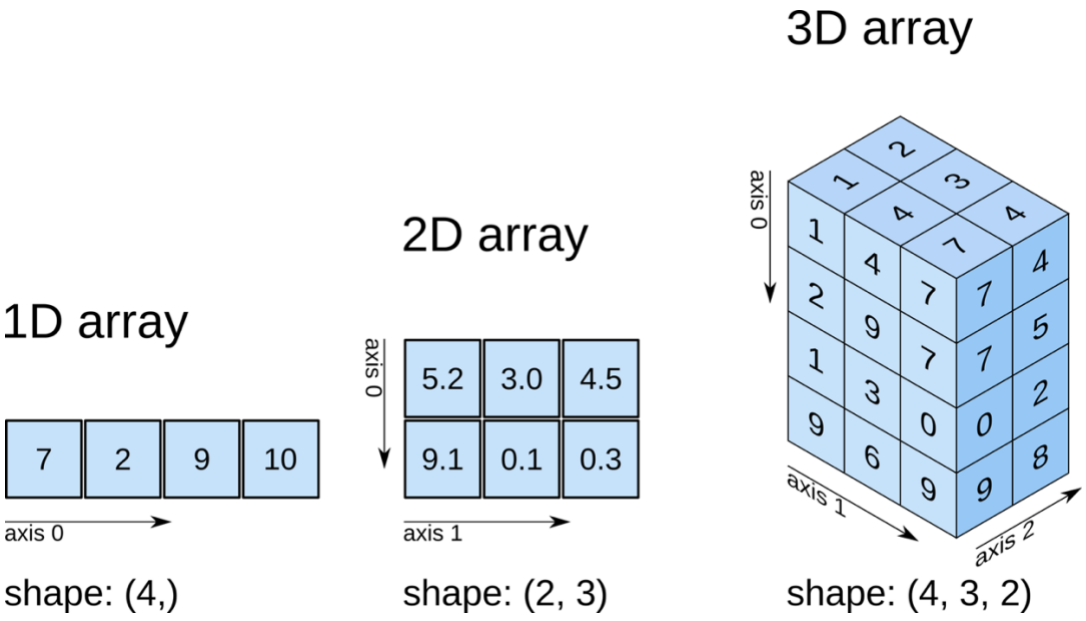
The NumPy library is a great alternative to python arrays. The difference is that the NumPy arrays are homogeneous that makes it easier to work with.

The NumPy arrays are convenient as they have the following **three features**–

- **Less Memory Requirement**
- **Faster Processing**
- **Convenience of use**

Besides their general usefulness, numpy arrays are also important to understand as they are used as the backbone for many other popular libraries. Modules such as SciPy, Matplotlib, pandas, scitkit-learn, scikit-image, and Biopython are all built on top of numpy arrays.

Example of numpy arrays:



Creating numpy arrays

Having learned how Python lists work, we can extend this to NumPy arrays. We have the option of creating N-dimensional arrays which are called ndarray.

Now, unlike lists, numpy arrays can only have one kind of data type (int, float, strings, etc). You can convert a python list to a NumPy array using the np.array() function.

```
>>> import numpy as np
>>> data = np.array([1,3,4,2,0])
>>> type(data)
<class 'numpy.ndarray'>
```

There are three common attributes of an ndarray that you should be aware of:

ndim – returns the array dimension (e.g. 1D, 2D, or 3D from the figure above)

```
>>> data.ndim
1
```

shape - tells us the size of the array in each dimension

```
>>> data.shape
(5,)
```

dtype – tells us the datatype of each of the elements in the array

```
>>> data.dtype
dtype('int64')
```

The following table lists all of the different types of datatypes used in numpy. While you should recognize the 4 main numerical classes we have already seen in Python (bool, int, float, and complex), numpy gives you a lot more control over the size of each datatype. This is important in some situations if you want to minimize the amount of storage space that is necessary to store a certain amount of data.

Data Type	Description
bool_	Boolean (True or False) stored as a byte
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2.15E-9 to 2.15E+9)
int64	Integer (-9.22E-18 to 9.22E+18)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4.29E+9)
uint64	Unsigned integer (0 to 1.84E+19)
float16	Half precision signed float
float32	Single precision signed float
float64	Double precision signed float
complex64	Complex number: two 32-bit floats (real and imaginary components)
complex128	Complex number: two 64-bit floats (real and imaginary components)

For example, there are four sizes of integers, int8, int16, int32, and int64. int8 takes one byte of data (or 8 bits), but can only store numbers between -128 to 127. If you know all your data falls within this range, e.g. you are storing the number of hours you spend sleeping each night or the number of miles you ran each day of the year, rounded to the nearest integer, you can use an int8 to store this data without issue. But if you are storing a larger number, such as the number of days per year that it rained in a specific location, you would need use a large dtype like int16.

You can also create an integer that can't be negative. The extra space typically used to store the +/- information expands the maximum value. So the largest value of an int8 is 127 while the largest value of an uint8 is 255.

You can specify the datatype when you create the array using the dtype argument or you can change the dtype of an existing array using the astype() method:

```
>>> np.array([1,3,4,2,0], dtype = 'float128')
array([1., 3., 4., 2., 0.], dtype=float128)
>>> np.array([1,3,4,2,0]).astype('float128')
array([1., 3., 4., 2., 0.], dtype=float128)
```

Creating multidimensional arrays

Multidimensional arrays can be created by lists of lists. In general, this is reasonable for creating small arrays but becomes rather unwieldy as the size of the data grows:

```
>>> data2 = np.array([[2,3,4],[1,5,6]])
>>> data3 = np.array([[[1,1],[2,2]],[[3,3],[4,4]]])
>>> data2
```

```
array([[2, 3, 4],
       [1, 5, 6]])
>>> data3
array([[[1, 1],
        [2, 2]],

       [[3, 3],
        [4, 4]]])
>>> data2.shape
(2, 3)
>>> data3.shape
(2, 2, 2)
```

Creating arrays filled with specific values

It is also easy to create an array of zeros of whatever shape and datatype you would like using np.zeros function. The individual elements can be modified afterword

```
>>> np.zeros(shape = (3,3,4), dtype = 'float128')
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]], dtype=float128)
```

Similarly you can also create arrays of ones:

```
>>> np.ones(shape = (2,2), dtype = 'int')
array([[1, 1],
       [1, 1]])
```

Or any value you would like:

```
>>> np.full(shape = (3,3), fill_value = 6, dtype = 'int')
array([[6, 6, 6],
       [6, 6, 6],
       [6, 6, 6]])
```

Accessing data in numpy arrays

Accessing specific elements

Like lists, individual elements can be accessed using the slicing operator.

```
>>> data
array([6, 3, 4, 3, 2])
>>> data[0]
6
>>> data[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 5 is out of bounds for axis 0 with size 5
>>> data[4]
2
>>> data[-1]
2
```

You can also modify specific elements this way:

```
>>> data[-1] = 1
>>> data
array([6, 3, 4, 3, 1])
```

Accessing specific elements in multidimensional arrays

```
>>> data2 = np.array([[2,3,4],[1,5,6]])
>>> data2
array([[2, 3, 4],
       [1, 5, 6]])
```

For multidimensional arrays, you need to specify an index for each dimension, separated by a comma. The first index specifies a row and the second index specifies a column.

```
>>> data2[0,1]
3
>>> data2[1,1]
5
```

If you try to access data that doesn't exist, you will get an index error

```
>>> data2[2,1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 2 is out of bounds for axis 0 with size 2
```

Creating smaller arrays from existing arrays

Using slicing to create smaller arrays

Similar to lists, you can also provide ranges of indices to specify which part of the array you would like

```
>>> data
array([6, 3, 4, 3, 1])
>>> data[1:3]
array([3, 4])
>>> data[1:]
array([3, 4, 3, 1])
>>> data2
array([[2, 3, 4],
       [1, 5, 6]])
>>> data2[0:2,0:2]
array([[2, 3],
       [1, 5]])
>>> data2[0:2,1:]
array([[3, 4],
       [5, 6]])
>>> data3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'data3' is not defined
```

```
>>> data2[0]
array([2, 3, 4])
>>> data2[0,:2]
array([2, 3])
```

Using filtering to create smaller arrays

Similar to pandas, you can use conditionals to filter data based upon the value of the element. Using conditionals on arrays returns an array of Booleans with the same shape as the original array. For example:

```
>>> data
array([6, 3, 4, 3, 1])
>>> data > 3
array([ True, False,  True, False, False])
>>> data2
array([[2, 3, 4],
       [1, 5, 6]])
>>> data2 > 3
array([[False, False,  True],
       [False,  True,  True]])
```

Again, like pandas, you can use this Boolean array with the slicing operator to return a subset of data:

```
>>> data[data > 3]
array([6, 4])
>>> data2[data2 > 3]
array([4, 5, 6])
```

Interestingly, you can modify these values however you like:

```
>>> data[data > 3] = -100
>>> data
array([-100,    3, -100,    3,    1])
```

Analyzing data in numpy arrays

Element by element manipulations

Numpy is designed to make array math very easy. For example, you might want to perform some sort of math on each element of the array. Many of the operators are customized to make this easy. For example, addition works like this:

```
>>> data
array([-100,    3, -100,    3,    1])
```

```
>>> data + 2
array([-98,   5, -98,   5,   3])
```

Note that each element had two added to it. Subtraction, ultiplication and division work very similarly:

```
>>> data - 2
array([-102,    1, -102,    1,   -1])
>>> data * 2
array([-200,    6, -200,    6,    2])
>>> data / 2
array([-50. ,   1.5, -50. ,   1.5,   0.5])
```

In each case, an array of identical shape is returned with each element modified.

Numpy also has a large number of built in functions that perform operations on each element. For example, if you wanted to take the sine of each number:

```
>>> np.sin(data)
array([0.50636564, 0.14112001, 0.50636564, 0.14112001, 0.84147098])
```

There are way too many functions available to go over individually, but a list of these functions can be found here:

https://numpy.org/doc/stable/reference/routines.math.html

Operators that act on two arrays work very similar. Each element of the arrays are added together. For example:

```
>>> data
array([-100,    3, -100,    3,    1])
>>> data_m = data * 2
>>> data
array([-100,    3, -100,    3,    1])
>>> data_m
array([-200,    6, -200,    6,    2])
>>> data + data_m
array([-300,    9, -300,    9,    3])
>>> data - data_m
array([100,  -3, 100,  -3,  -1])
>>> data * data_m
array([20000,    18, 20000,    18,     2])
>>> data / data_m
array([0.5, 0.5, 0.5, 0.5, 0.5])
```

Summary functions

Numpy also has a large number of functions to perform statistical analysis of your data. For example, you can very easily calculate the mean of an array:

```
>>> data
array([-100,    3, -100,    3,    1])
>>> data.mean()
-38.6
```

Or the standard deviation:

```
>>> data.std()
50.13820898277081
```

Again, there are a large number of statistical functions available for you to use, which can be found here:

https://numpy.org/doc/stable/reference/routines.statistics.html

Finally, summary functions on multidimensional arrays are possible but a little bit tricky when you first start to use them.

```
>>> data2
array([[2, 3, 4],
       [1, 5, 6]])
>>> data2.mean()
3.5
>>> data2.std()
1.707825127659933
```

Mostly straight forward if you want to calculate the mean of the entire array. But what if you only want to calculate the mean of each row or each column? This is possible using the axis argument:

```
>>> data2.mean(axis = 0)
array([1.5, 4. , 5. ])
>>> data2.mean(axis = 1)
array([3., 4.])
```

Note that the you can specify which dimension to group the data by. For 3D arrays you can also summarize data over 2D slices of the data:

```
>>> data3
array([[[1, 1],
        [2, 2]],

       [[3, 3],
        [4, 4]]])
```

```
>>> data3.mean()
2.5
>>> data3.mean(axis = 0)
array([[2., 2.],
       [3., 3.]])
>>> data3.mean(axis = (0,1))
array([2.5, 2.5])
```