Python NumPy Library for Data Analysis

NumPy (Numerical Python) is a Python library for data science and numerical computing. Many advanced data science and machine learning libraries require data to be in the form of NumPy arrays before it can be processed. In this chapter, you are going to learn some of the most commonly used functionalities of the NumPy array. NumPy comes prebuilt with Anaconda's distribution of Python. Or else, you can install NumPy with the following pip command in a terminal or a command prompt:

```
$ pip install numpy
```

3.1. Advantages of NumPy Library

A NumPy array has many advantages over regular Python lists. Some of them are listed below:

1. NumPy arrays are much faster for insertion, deletion, updating, and reading
2. NumPy arrays contain advanced broadcasting functionalities compared with Python arrays.
3. NumPy array comes with a lot of methods that support advanced arithmetic linear algebra options.
4. NumPy provides advanced multi-dimensional array slicing capabilities.

In the next section, you will see how to create NumPy arrays using different methods.

3.2. Creating NumPy Arrays

Depending upon the type of data you need inside your NumPy array, different methods can be used to create a NumPy array.

3.2.1. Using Array Methods

To create a NumPy array, you can pass a list to the **array()** method of the NumPy module as shown below:

Script 1:

```
1. import numpy as np
2. nums_list = [10,12,14,16,20]
3. nums_array = np.array(nums_list)
4. type(nums_array)
```

Output:

```
numpy.ndarray
```

You can also create a multi-dimensional NumPy array. To do so, you need to create a list of lists where each internal list corresponds to the row in a 2-dimensional array. Here is an example of how to create a 2-dimensional array using the **array()** method.

Script 2:

```
1. row1 = [10,12,13]
2. row2 = [45,32,16]
3. row3 = [45,32,16]
4.
5. nums_2d = np.array([row1, row2, row3])
6. nums_2d.shape
```

Output:

```
(3, 3)
```

3.2.2. Using Arrange Method

With the **arrange** () method, you can create a NumPy array that contains a range of integers. The first parameter to the arrange method is the lower bound, and the second parameter is the upper bound. The lower bound is included in the array. However, the upper bound is not included. The following script creates a NumPy array with integers 5 to 10.

Script 3:

```
1. nums_arr = np.arange(5,11)
2. print (nums_arr)
```

Output:

```
[5 6 7 8 9 10]
```

You can also specify the step as a third parameter in the **arrange()** function. A step defines the distance between two consecutive points in the array. The following script creates a NumPy array from 5 to 11 with a step size of 2.

Script 4:

```
1. nums_arr = np.arange(5,12,2)
2. print (nums_arr)
```

Output:

```
[5 7 9 11]
```

3.2.3. Using Ones Method

The **ones()** method can be used to create a NumPy array of all ones. Here is an example.

Script 5:

```
1. ones_array = np.ones(6)
```

```
2. print (ones_array)
```

Output:

```
[1. 1. 1. 1. 1. 1.]
```

You can create a 2-dimensional array of all ones by passing the number of rows and columns as the first and second parameters of the **ones()** method, as shown below:

Script 6:

```
1. ones_array = np.ones((6,4))
2. print (ones_array)
```

Output:

```
[[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]]
```

3.2.4. Using Zeros Method

The **zeros()** method can be used to create a NumPy array of all zeros. Here is an example.

Script 7:

```
1. zeros_array = np.zeros(6)
2. print (zeros_array)
```

Output:

```
[0. 0. 0. 0. 0. 0.]
```

You can create a 2-dimensional array of all zeros by passing the number of rows and columns as the first and second parameters of the **zeros()** method as shown below:

Script 8:

```
1. zeros_array = np.zeros((6,4))
2. print (zeros_array)
```

Output:

```
[[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]
```

3.2.5. using Eyes Method

The **eye()** method is used to create an identity matrix in the form of a 2-dimensional numPy array. An identity contains 1s along the diagonal, while the rest of the elements are 0 in the array.

Script 9:

```
1. eyes_array = np.eye(5)
2. print (eyes_array)
```

Output:

```
[[1. 0. 0. 0. 0.]
[0. 1. 0. 0. 0.]
[0. 0. 1. 0. 0.]
[0. 0. 0. 1. 0.]
[0. 0. 0. 0. 1.]]
```

3.2.6. Using Random Method

The **random.rand()** function from the NumPy module can be used to create a NumPy array with uniform distribution.

Script 10:

```
1. uniform_random = np.random.rand(4, 5)
2. print (uniform_random)
```

Output:

```
[[0.36728531 0.25376281 0.05039624 0.96432236 0.08579293]
[0.29194804 0.93016399 0.88781312 0.50209692 0.63069239]
[0.99952044 0.44384871 0.46041845 0.10246553 0.53461098]
[0.75817916 0.36505441 0.01683344 0.9887365 0.21490949]]
```

The **random.randn()** function from the NumPy module can be used to create a NumPy array with normal distribution, as shown in the following example.

Script 11:

```
1. normal_random = np.random.randn(4, 5)
2. print (uniform_random)
```

Output:

```
[[0.36728531 0.25376281 0.05039624 0.96432236 0.08579293]
[0.29194804 0.93016399 0.88781312 0.50209692 0.63069239]
[0.99952044 0.44384871 0.46041845 0.10246553 0.53461098]
[0.75817916 0.36505441 0.01683344 0.9887365 0.21490949]]
```

Finally, the **random.randint()** function from the NumPy module can be used to create a NumPy array with random integers between a certain range. The first parameter to the **randint()** function specifies the lower bound, the second parameter specifies the upper bound, while the last parameter specifies the number of random integers to generate between the range. The following example generates five random integers between 5 and 50.

Script 12:

```
1. integer_random = np.random.randint(10, 50, 5)
2. print (integer_random)
```

Output:

```
[25 49 21 35 17]
```

3.3. Reshaping NumPy Arrays

A NumPy array can be reshaped using the **reshape()** function. It is important to mention that the product of the rows and columns in the reshaped array must be equal to the product of rows and columns in the original array. For instance, in the following example, the original array contains four rows and six columns, i.e., 4 x 6 = 24. The reshaped array contains three rows and eight columns, i.e., 3 x 8 = 24.

Script 13:

```
1. uniform_random = np.random.rand(4, 6)
2. uniform_random = uniform_random.reshape(3, 8)
3. print (uniform_random)
```

Output:

```
[[0.37576967 0.5425328 0.56087883 0.35265748 0.19677258 0.65107479 0.63287089 0.70649913]
[0.47830882 0.3570451 0.82151482 0.09622735 0.1269332 0.65866216 0.31875221 0.91781242]
[0.89785438 0.47306848 0.58350797 0.4604004 0.62352155 0.88064432 0.0859386 0.51918485]]
```

3.4. Array Indexing And Slicing

NumPy arrays can be indexed and sliced. Slicing an array means dividing an array into multiple parts.

NumPy arrays are indexed just like normal lists. Indexes in NumPy arrays start from 0, which means that the first item of a NumPy array is stored at the 0^{th} index.

The following script creates a simple NumPy array of the first 10 positive integers.

Script 14:

```
1. s = np.arange(1,11)
2. print (s)
```

Output:

```
[ 1 2 3 4 5 6 7 8 9 10]
```

The item at index one can be accessed as follows:

Script 15:

```
print (s[1])
```

Output:

```
2
```

To slice an array, you have to pass the lower index, followed by a colon and the upper index. The items from the lower index (inclusive) to the upper index (exclusive) will be filtered. The following script slices the array "s" from the 1st index to the 9th index. The elements from index 1 to 8 are printed in the output.

Script 16:

```
print (s[1:9])
```

Output:

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

if you specify only the upper bound, all the items from the first index to the upper bound are returned. similarly, if you specify only the lower bound, all the items from the lower bound to the last item of the array are returned.

Script 17:

```
1. print (s[:5])
2. print (s[5:])
```

Output:

```
[1 2 3 4 5]
[ 6 7 8 9 10]
```

Array slicing can also be applied on a 2-dimensional array. To do so, you have to apply slicing on arrays and columns separately. A comma separates the rows and columns slicing. In the following script, the rows from the first and second index are returned, While all the columns returned. You can see the first two complete rows in the output.

Script 18:

```
1. row1 = [10,12,13]
2. row2 = [45,32,16]
3. row3 = [45,32,16]
4.
5. nums_2d = np.array([row1, row2, row3])
6. print (nums_2d[:2,:])
```

Output:

```
[[10 12 13]
[45 32 16]]
```

Similarly, the following script returns all the rows but only the first two columns.

Script 19:

```
1. row1 = [10,12,13]
2. row2 = [45,32,16]
3. row3 = [45,32,16]
4.
5. nums_2d = np.array([row1, row2, row3])
```

```
6. print (nums_2d[:,:2])
```

Output:

```
[[10 12]
[45 32]
[45 32]]
```

Let's see another example of slicing. Here, we will slice the rows from row one to the end of rows and column one to the end of columns. (Remember, row and column numbers start from 0.) In the output, you will see the last two rows and the last two columns.

Script 20:

```
1. row1 = [10,12,13]
2. row2 = [45,32,16]
3. row3 = [45,32,16]
4.
5. nums_2d = np.array([row1, row2, row3])
6. print (nums_2d[1:,1:])
```

Output:

```
[[32 16]
[32 16]]
```

3.5. NumPy for Arithmetic Operations

NumPy arrays provide a variety of functions to perform arithmetic operations. Some of these functions are explained in this section.

3.5.1. Finding Square Roots

The **sqrt()** function is used to find the square roots of all the elements in a list as shown below:

Script 21:

```
1. nums = [10,20,30,40,50]
2. np_sqr = np.sqrt(nums)
3. print (np_sqr)
```

Output:

```
[3.16227766 4.47213595 5.47722558 6.32455532 7.07106781]
```

3.5.2. Finding Logs

The **log()** function is used to find the logs of all the elements in a list as shown below:

Script 22:

```
1. nums = [10,20,30,40,50]
2. np_log = np.log(nums)
3. print (np_log)
```

Output:

```
[2.30258509 2.99573227 3.40119738 3.68887945 3.91202301]
```

3.5.3. Finding Exponents

The **exp()** function takes the exponents of all the elements in a list as shown below:

Script 23:

```
1. nums = [10,20,30,40,50]
2. np_exp = np.exp(nums)
3. print (np_exp)
```

Output:

```
[2.20264658e+04 4.85165195e+08 1.06864746e+13 2.35385267e+17 5.18470553e+21]
```

3.5.4. Finding Sine and Cosine

You can find the sines and cosines of items in a list using the sine and cosine function, respectively, as shown in the following script.

Script 24:

```
1. nums = [10,20,30,40,50]
2. np_sine = np.sin(nums)
3. print (np_sine)
4.
5. nums = [10,20,30,40,50]
6. np_cos = np.cos(nums)
7. print (np_cos)
```

Output:

```
[-0.54402111 0.91294525 -0.98803162 0.74511316 -0.26237485]
[-0.83907153 0.40808206 0.15425145 -0.66693806 0.96496603]
```

3.6. NumPy for Linear Algebra Operations

Data science makes extensive use of linear algebra. The support for performing advanced linear algebra functions in a fast and efficient way makes NumPy one of the most routinely used libraries for data science. In this section, you will perform some of the most linear algebraic operations with NumPy.

3.6.1. Finding Matrix Dot Product

To find a matrix dot product, you can use the **dot()** function. To find the dot product, the number of columns in the first matrix must match the number of rows in the second matrix. Here is an example.

Script 25:

```
1. A = np.random.randn(4,5)
2.
3. B = np.random.randn(5,4)
4.
5. Z = np.dot(A,B)
6.
7. print (Z)
```

Output:

```
[[ 1.43837722 -4.74991285 1.42127048 -0.41569506]
[-1.64613809 5.79380984 -1.33542482 1.53201023]
[-1.31518878 0.72397674 -2.01300047 0.61651047]
[-1.36765444 3.83694475 -0.56382045 0.21757162]]
```

3.6.2. Element-wise Matrix Multiplication

In addition to finding the dot product of two matrices, you can element-wise multiply two matrices. To do so, you can use the **multiply()** function. The dimensions of the two matrices must match.

Script 26:

```
1. row1 = [10,12,13]
2. row2 = [45,32,16]
3. row3 = [45,32,16]
4.
5. nums_2d = np.array([row1, row2, row3])
6. multiply = np.multiply(nums_2d, nums_2d)
7. print (multiply)
```

Output:

```
[[ 100 144 169]
[2025 1024 256]
[2025 1024 256]]
```

3.6.3. Finding Matrix Inverse

You find the inverse of a matrix via the **linalg.inv()** function as shown below:

Script 27:

```
1. row1 = [1,2,3]
2. row2 = [4,5,6]
3. row3 = [7,8,9]
4.
5. nums_2d = np.array([row1, row2, row3])
```

```
6.
7. inverse = np.linalg.inv(nums_2d)
8. print (inverse)
```

Output:

```
[[ 3.15251974e+15 -6.30503948e+15 3.15251974e+15]
[-6.30503948e+15 1.26100790e+16 -6.30503948e+15]
[ 3.15251974e+15 -6.30503948e+15 3.15251974e+15]]
```

3.6.4. Finding Matrix Determinant

Similarly, the determinant of a matrix can be found using the **linalg.det()** function as shown below:

Script 28:

```
1. row1 = [1,2,3]
2. row2 = [4,5,6]
3. row3 = [7,8,9] 4.
5. nums_2d = np.array([row1, row2, row3])
6.
7. determinant = np.linalg.det(nums_2d)
8. print (determinant)
```

Output:

```
-9.51619735392994e-16
```

3.6.5. Finding Matrix Trace

The trace of a matrix refers to the sum of all the elements along the diagonal of a matrix. To find the trace of a matrix, you can use the **trace()** function, as shown below:

Script 29:

```
1. row1 = [1,2,3]
2. row2 = [4,5,6]
3. row3 = [7,8,9]
```

```
4.
5. nums_2d = np.array([row1, row2, row3])
6.
7. trace = np.trace(nums_2d)
8. print (trace)
```

Output:

```
15
```