NumPy Arrays

NumPy and SciPy are open source add-on modules for Python that provide common mathematical and numerical routines in pre-compiled, fast functions. Over the years, these have grown into highly mature libraries that provide functionality that meets, or perhaps exceeds, what is associated with common commercial software such as Matlab or Mathematica.

NumPy arrays are different than common Python lists since Python lists can be thought of as simple arrays. NumPy arrays are built for mathematical vectorized operations that process a lot of numerical data with just a single line of code. Many built-in mathematical functions in NumPy arrays are written in low-level languages such as C or Fortran and are pre-compiled for really fast execution

NumPy arrays are optimized data structures for numerical analysis, and that's why they are so important to data scientists.

Exercise 1: Creating a NumPy Array (from a List)

In this exercise, we will create a `NumPy` array from a list. We're going to define a list first and use the array function of the `NumPy` library to convert the list into an array. Next, we'll read from a `.csv` file and store the data in a `NumPy` array using the `genfromtxt` function of the `NumPy` library. To do so, let's go through the following steps:

1. To work with **NumPy**, we must import it. By convention, we give it a short name, np, while importing it. This will make referencing the objects under the **NumPy** package organized:

   ```
   import numpy as np
   ```
2. Create a list with three elements: **1**, **2**, and **3**:

   ```
   list_1 = [1,2,3]
   list_1
   ```

The output is as follows:

   ```
   [1, 2, 3]
   ```
3. Use the **array** function to convert it into an array:

   ```
   array_1 = np.array(list_1)
   array_1
   ```

The output is as follows:

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

We just created a **NumPy** array object called **array_1** from the regular Python list object, **list_1**.

4. Create an array of floating type elements, that is, **1.2**, **3.4**, and **5.6**, using the array function directly:

   ```
   a = np.array([1.2, 3.4, 5.6])
   a
   ```

The output is as follows:

   ```
   array([1.2, 3.4, 5.6])
   ```
5. Let's check the type of the newly created object, a, using the **type** function:

   ```
   type(a)
   ```

The output is as follows:

   ```
   numpy.ndarray
   ```
6. Use the **type** function to check the type of **array_1**:

   ```
   type(array_1)
   ```

The output is as follows:

```
numpy.ndarray
```

As we can see, both a and **array_1** are **NumPy** arrays.

7. Now, use type on **list_1**:

```
type(list_1)
```

The output is as follows:

```
list
```

As we can see, **list_1** is essentially a Python list and we have used the array function of the **NumPy** library to create a **NumPy** array from that list.

8. Now, let's read a **.csv** file as a **NumPy** array using the **genfromtxt** function of the **NumPy** library:

```
data = np.genfromtxt('../datasets/stock.csv', \
                     delimiter=',',names=True,dtype=None, \
                     encoding='ascii')
data
```

Note

The path (highlighted) should be specified based on the location of the file on your system.

The partial output is as follows:

```
array([('MMM', 100), ('AOS', 101), ('ABT', 102), ('ABBV', 103),
       ('ACN', 104), ('ATVI', 105), ('AYI', 106), ('ADBE', 107),
       ('AAP', 108), ('AMD', 109), ('AES', 110), ('AET', 111),
       ('AMG', 112), ('AFL', 113), ('A', 114), ('APD', 115),
       ('AKAM', 116), ('ALK', 117), ('ALB', 118), ('ARE', 119),
       ('ALXN', 120), ('ALGN', 121), ('ALLE',
122), ('AGN', 123),
       ('ADS', 124), ('LNT', 125), ('ALL',
126), ('GOOGL', 127),
       ('GOOG', 128), ('MO', 129), ('AMZN',
130), ('AEE', 131),
       ('AAL', 132), ('AEP', 133), ('AXP',
134), ('AIG', 135),
       ('AMT', 136), ('AWK', 137), ('AMP',
138), ('ABC', 139),
       ('AME', 140), ('AMGN', 141), ('APH',
142), ('APC', 143),
       ('ADI', 144), ('ANDV', 145), ('ANSS',
146), ('ANTM', 147),
       ('AON', 148)], dtype=[('Symbol',
'<U5'), ('Price', '<i8')])
```

9. Use the **`type`** function to check the type of **`data`**:

```
type(data)
```

The output is as follows:

```
numpy.ndarray
```

As we can see, the data variable is also a **`NumPy`** array.

From this exercise, we can observe that the **`NumPy`** array is different from the regular list object. The most important point to keep in mind is that **`NumPy`** arrays do not have the same methods as lists and that they are essentially designed for mathematical functions.

`NumPy` arrays are like mathematical objects – **vectors**. They are built for element-wise operations, that is, when we add two **`NumPy`** arrays, we add the first element of the first array to the first element of the second array – there is an element-to-element correspondence in this operation.

This is in contrast to Python lists, where the elements are simply appended and there is no element-to-element relation. This is the real power of a NumPy array: they can be treated just like mathematical vectors.

With this knowledge, we're going to perform the addition operation on **`NumPy`** arrays in the next exercise.

Exercise 2: Adding Two NumPy Arrays

This simple exercise will demonstrate the addition of two **`NumPy`** arrays using the **+** notation, and thereby show the key difference between a regular Python list/array and a **`NumPy`** array. Let's perform the following steps:

1. Import the **`NumPy`** library:
   ```
   import numpy as np
   ```
2. Declare a Python list called **`list_1`** and a **`NumPy`** array:
   ```
   list_1 = [1,2,3]
   array_1 = np.array(list_1)
   ```
3. Use the **+** notation to concatenate two **`list_1`** objects and save the results in **`list_2`**:
   ```
   list_2 = list_1 + list_1
   list_2
   ```
   The output is as follows:
   ```
   [1, 2, 3, 1, 2, 3]
   ```
4. Use the same **+** notation to concatenate two **`array_1`** objects and save the result in **`array_2`**:
   ```
   array_2 = array_1 + array_1
   array_2
   ```
   The output is as follows:

```
[2 ,4, 6]
```

5. Load a **`.csv`** file and concatenate it with itself:

```
data =
np.genfromtxt('../datasets/numbers.csv', \
                     delimiter=',',
names=True)
data = data.astype('float64')
data + data
```

Note

The path (highlighted) should be specified based on the location of the file on your system. The **`.csv`** *file that will be used is* **`numbers.csv`**

The output is as follows:

```
array([202., 204., 206., 208., 210., 212.,
214., 216., 218.,
       220., 222., 224., 226., 228., 230.,
232., 234., 236.,
       238., 240., 242., 244., 246., 248.,
250., 252., 254.,
       256., 258., 260., 262., 264., 266.,
268., 270., 272.,
       274., 276., 278., 280., 282., 284.,
286., 288., 290.,
       292., 294., 296.])
```

Did you notice the difference? The first **print** shows a list with **`6`** elements, [**`1, 2, 3, 1, 2, 3`**], but the second **print** shows another **`NumPy`** array (or vector) with the elements [**`2, 4, 6`**], which are just the sum of the individual elements of **`array_1`**. As we discussed earlier, **`NumPy`** arrays are perfectly designed to perform element-wise operations since there is element-to-element correspondence.

NumPy arrays even support element-wise exponentiation. For example, suppose there are two arrays – the elements of the first array will be raised to the power of the elements in the second array.

In the following exercise, we will try out some mathematical operations on **NumPy** arrays.

Exercise 3: Mathematical Operations on NumPy Arrays

In this exercise, we'll generate a **NumPy** array with the values extracted from a **.csv** file. We'll be using the multiplication and division operators on the generated **NumPy** array. Let's go through the following steps:

Note

The **.csv** *file that will be used is* **numbers.csv**.

Import the **NumPy** library and create a **NumPy** array from the **.csv** file:

```
import numpy as np
data = 
np.genfromtxt('../datasets/numbers.csv', \
                     delimiter=',', 
names=True)
data = data.astype('float64')
data
```

Note

Don't forget to change the path (highlighted) based on the location of the file on your system.

The output is as follows:

```
array([101., 102., 103., 104., 105., 106.,
107., 108., 109.,
       110., 111., 112., 113., 114., 115.,
116., 117., 118.,
       119., 120., 121., 122., 123., 124.,
125., 126., 127.,
       128., 129., 130., 131., 132., 133.,
134., 135., 136.,
       137., 138., 139., 140., 141., 142.,
143., 144., 145.,
       146., 147., 148.])
```

1. Multiply **45** by every element in the array:

```
data * 45
```

The output is as follows:

```
array([4545., 4590., 4635., 4680., 4725.,
4770., 4815., 4860.,
       4905., 4950., 4995., 5040., 5085.,
5130., 5175., 5220.,
       5265., 5310., 5355., 5400., 5445.,
5490., 5535., 5580.,
       5625., 5670., 5715., 5760., 5805.,
5850., 5895., 5940.,
       5985., 6030., 6075., 6120., 6165.,
6210., 6255., 6300.,
       6345., 6390., 6435., 6480., 6525.,
6570., 6615., 6660.])
```

2. Divide the array by **67.7**:

```
data / 67.7
```

The output is as follows:

```
array([1.49187592, 1.50664697, 1.52141802,
1.53618907,
       1.55096012, 1.56573117, 1.58050222,
1.59527326,
       1.61004431, 1.62481536, 1.63958641,
1.65435746,
       1.66912851, 1.68389956, 1.69867061,
1.71344165,
       1.7282127 , 1.74298375, 1.7577548 ,
1.77252585,
       1.7872969 , 1.80206795, 1.816839 ,
1.83161004,
       1.84638109, 1.86115214, 1.87592319,
1.89069424,
       1.90546529, 1.92023634, 1.93500739,
1.94977843,
       1.96454948, 1.97932053, 1.99409158,
2.00886263,
       2.02363368, 2.03840473, 2.05317578,
2.06794682,
       2.08271787, 2.09748892, 2.11225997,
2.12703102,
       2.14180207, 2.15657312, 2.17134417,
2.18611521])
```

3. Raise one array to the second array's power using the following command:

```
list_1 = [1,2,3]
array_1 = np.array(list_1)
print("array_1 raised to the power of
array_1: ", \
      array_1**array_1)
```

The output is as follows:

```
array_1 raised to the power of array_1: [ 1 4
27]
```

Thus, we can observe how NumPy arrays allow element-wise exponentiation.

In the next section, we'll discuss how to apply advanced mathematical operations to NumPy arrays.

Advanced Mathematical Operations

Generating numerical arrays is a fairly common task. So far, we have been doing this by creating a Python list object and then converting that into a **`NumPy`** array. However, we can bypass that and work directly with native NumPy methods. The **`arange`** function creates a series of numbers based on the minimum and maximum bounds you give and the step size you specify. Another function, **`linspace`**, creates a series of fixed numbers of the intermediate points between two extremes.

In the next exercise, we are going to create a list and then convert that into a **`NumPy`** array. We will then show you how to perform some advanced mathematical operations on that array.

Exercise 4: Advanced Mathematical Operations on NumPy Arrays

In this exercise, we'll practice using all the built-in mathematical functions of the **`NumPy`** library. Here, we are going to be creating a list and converting it into a **`NumPy`** array. Then, we will perform some advanced mathematical operations on that array. Let's go through the following steps:

Note

We're going to use the **`numbers.csv`** *file in this exercise.*

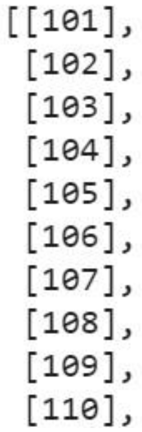
Import the **`pandas`** library and read from the **`numbers.csv`** file using **`pandas`**. Then, convert it into a list:

```
import pandas as pd
df = pd.read_csv("../datasets/numbers.csv")
list_5 = df.values.tolist()
list_5
```

Note

Don't forget to change the path (highlighted) based on the location of the file on your system.

The output (partially shown) is as follows:



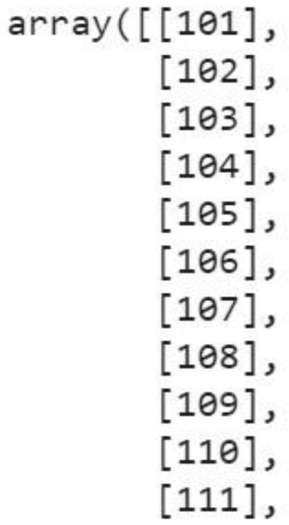
```
[[101],
 [102],
 [103],
 [104],
 [105],
 [106],
 [107],
 [108],
 [109],
 [110],
```

Figure 1: Partial output of the .csv file

1. Convert the list into a **NumPy** array by using the following command:

```
import numpy as np
array_5 = np.array(list_5)
array_5
```

The output (partially shown) is as follows:



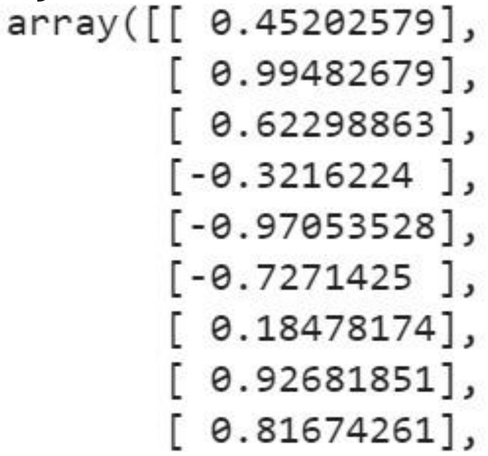
```
array([[101],
       [102],
       [103],
       [104],
       [105],
       [106],
       [107],
       [108],
       [109],
       [110],
       [111],
```

Figure 2: Partial output of the NumPy array

2. Find the sine value of the array by using the following command:

```
# sine function
np.sin(array_5)
```

The output (partially shown) is as follows:



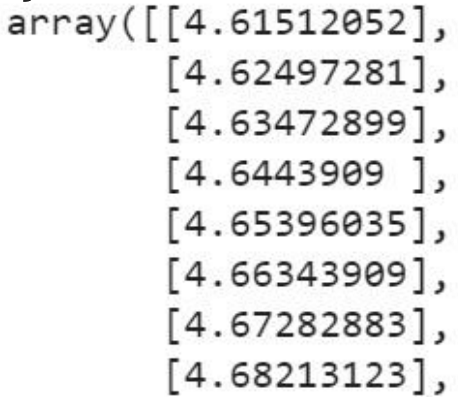
```
array([[ 0.45202579],
       [ 0.99482679],
       [ 0.62298863],
       [-0.3216224 ],
       [-0.97053528],
       [-0.7271425 ],
       [ 0.18478174],
       [ 0.92681851],
       [ 0.81674261],
```

Figure 3: Partial output of the sine value

3. Find the logarithmic value of the array by using the following command:

```
# logarithm
np.log(array_5)
```

The output (partially shown) is as follows:



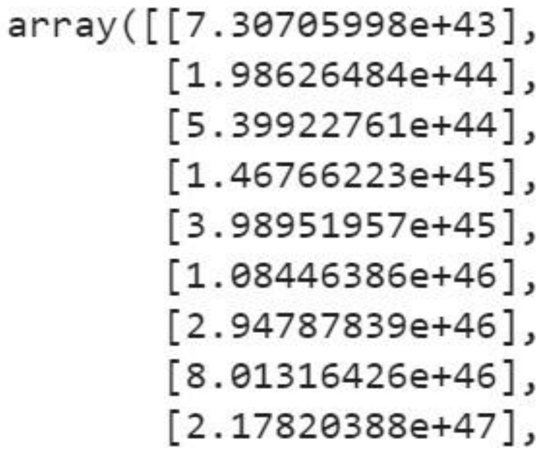
```
array([[4.61512052],
       [4.62497281],
       [4.63472899],
       [4.6443909 ],
       [4.65396035],
       [4.66343909],
       [4.67282883],
       [4.68213123],
```

Figure 4: Partial output of the logarithmic array

4. Find the exponential value of the array by using the following command:

```
# Exponential
np.exp(array_5)
```

The output (partially shown) is as follows:



```
array([[7.30705998e+43],
       [1.98626484e+44],
       [5.39922761e+44],
       [1.46766223e+45],
       [3.98951957e+45],
       [1.08446386e+46],
       [2.94787839e+46],
       [8.01316426e+46],
       [2.17820388e+47],
```

Figure 5: Partial output of the exponential array

As we can see, advanced mathematical operations are fairly easy to perform on a **`NumPy`** array using the built-in methods.

Exercise 5: Generating Arrays Using arange and linspace Methods

This exercise will demonstrate how we can create a series of numbers using the **`arange`** method. To make the list linearly spaced, we're going to use the **`linspace`** method. To do so, let's go through the following steps:

1. Import the **`NumPy`** library and create a series of numbers using the **`arange`** method using the following command:

   ```
   import numpy as np
   np.arange(5,16)
   ```

   The output is as follows:

   ```
   array([ 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
   ])
   ```

2. Print numbers using the **`arange`** function by using the following command:

   ```
   print("Numbers spaced apart by 2: ",\
   ```

```
      np.arange(0,11,2))
print("Numbers spaced apart by a floating
point number: ",\
      np.arange(0,11,2.5))
print("Every 5th number from 30 in reverse
order\n",\
      np.arange(30,-1,-5))
```

The output is as follows:

```
Numbers spaced apart by 2: [ 0 2 4 6 8 10]
Numbers spaced apart by a floating point
number:
[ 0. 2.5 5.0 7.5 10. ]
Every 5th number from 30 in reverse order
[30 25 20 15 10 5 0]
```

3. For linearly spaced numbers, we can use the **`linspace`** method, as follows:

```
print("11 linearly spaced numbers between 1
and 5: ",\
      np.linspace(1,5,11))
```

The output is as follows:

```
11 linearly spaced numbers between 1 and 5:
[1. 1.4 1.8 2.2 2.6 3. 3.4 3.8 4.2 4.6 5. ]
```

As we can see, the **`linspace`** method helps us in creating linearly spaced elements in an array.

So far, we have only created one-dimensional arrays. Now, let's create some multi-dimensional arrays (such as a matrix in linear algebra).

Exercise 6: Creating Multi-Dimensional Arrays

In this exercise, just like we created the one-dimensional array from a simple flat list, we will create a two-dimensional array from a list of lists.

Note

This exercise will use the **`numbers2.csv`** *file.*

Let's go through the following steps:

1. Import the necessary Python libraries, load the **`numbers2.csv`** file, and convert it into a two-dimensional **`NumPy`** array by using the following commands:

   ```
   import pandas as pd
   import numpy as np
   df = pd.read_csv("../datasets/numbers2.csv",\
                    header=None)
   list_2D = df.values
   mat1 = np.array(list_2D)
   print("Type/Class of this object:",\
         type(mat1))
   print("Here is the matrix\n----------\n",\
         mat1, "\n----------")
   ```

   Note

   Don't forget to change the path (highlighted) based on the location of the file on your system.

   The output is as follows:

   ```
   Type/Class of this object: <class
   'numpy.ndarray'>
   Here is the matrix
   ----------
   [[1 2 3]
   [4 5 6]
   [7 8 9]]
   ----------
   ```

2. Tuples can be converted into multi-dimensional arrays by using the following code:

   ```
   tuple_2D = np.array([(1.5,2,3), (4,5,6)])
   ```

```
mat_tuple = np.array(tuple_2D)
print (mat_tuple)
```

The output is as follows:

```
[[1.5 2.  3. ]
[4.  5.  6. ]]
```

Thus, we have created multi-dimensional arrays using Python lists and tuples.

Now, let's determine the dimension, shape, size, and data type of the two-dimensional array.

Exercise 7: The Dimension, Shape, Size, and Data Type of Two-dimensional Arrays

This exercise will demonstrate a few methods that will let you check the dimension, shape, and size of the array.

Note that if it's a `3x2` matrix, that is, it has `3` rows and `2` columns, then the shape will be (`3,2`), but the size will be `6`, as in `6 = 3x2`. To learn how to find out the dimensions of an array in Python, let's go through the following steps:

1. Import the necessary Python modules and load the **`numbers2.csv`** file:

   ```
   import pandas as pd
   import numpy as np
   df = pd.read_csv("../datasets/numbers2.csv",\
                    header=None)
   list_2D = df.values
   mat1 = np.array(list_2D)
   ```

   Note

   Don't forget to change the path (highlighted) based on the location of the file on your system.

2. Print the dimension of the matrix using the **ndim** function:

```
print("Dimension of this matrix: ",
mat1.ndim,sep='')
```

The output is as follows:

```
Dimension of this matrix: 2
```

3. Print the size using the **size** function:

```
print("Size of this matrix: ",
mat1.size,sep='')
```

The output is as follows:

```
Size of this matrix: 9
```

4. Print the shape of the matrix using the **shape** function:

```
print("Shape of this matrix: ",
mat1.shape,sep='')
```

The output is as follows:

```
Shape of this matrix: (3, 3)
```

5. Print the dimension type using the **dtype** function:

```
print("Data type of this matrix: ",
mat1.dtype,sep='')
```

The output is as follows:

```
Data type of this matrix: int64
```

In this exercise, we looked at the various utility methods available in order to check the dimensions of an array. We used the **dnim**, **shape**, **dtype**, and **size** functions to look at the dimension of the array.

Now that we are familiar with basic vector (one-dimensional) and matrix data structures in NumPy, we will be able to create special matrices with ease. Often, you may have to create matrices filled with zeros, ones, random numbers, or ones in a diagonal fashion. An identity matrix is a matrix filled with zeros and ones in a diagonal from left to right.

Exercise 8: Zeros, Ones, Random, Identity Matrices, and Vectors

In this exercise, we will be creating a vector of zeros and a matrix of zeros using the **zeros** function of the **NumPy** library. Then, we'll create a matrix of fives using the **ones** function, followed by generating an identity matrix using the **eye** function. We will also work with the **random** function, where we'll create a matrix filled with random values. To do this, let's go through the following steps:

1. Print the vector of zeros by using the following command:
   ```
   import numpy as np
   print("Vector of zeros: ",np.zeros(5))
   ```
   The output is as follows:
   ```
   Vector of zeros: [0. 0. 0. 0. 0.]
   ```
2. Print the matrix of zeros by using the following command:
   ```
   print("Matrix of zeros: ",np.zeros((3,4)))
   ```
   The output is as follows:
   ```
   Matrix of zeros: [[0. 0. 0. 0.]
    [0. 0. 0. 0.]
    [0. 0. 0. 0.]]
   ```
3. Print the matrix of fives by using the following command:
   ```
   print("Matrix of 5's: ",5*np.ones((3,3)))
   ```
   The output is as follows:
   ```
   Matrix of 5's: [[5. 5. 5.]
   ```

```
 [5. 5. 5.]
 [5. 5. 5.]]
```

4. Print an identity matrix by using the following command:

```
print("Identity matrix of dimension
2:",np.eye(2))
```

The output is as follows:

```
Identity matrix of dimension 2: [[1. 0.]
 [0. 1.]]
```

5. Print an identity matrix with a dimension of **`4x4`** by using the following command:

```
print("Identity matrix of dimension
4:",np.eye(4))
```

The output is as follows:

```
Identity matrix of dimension 4: [[1. 0. 0.
0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

6. Print a matrix of random shape using the **`randint`** function:

```
print("Random matrix of shape(4,3):\n",\
      np.random.randint(low=1,high=10,size=(4
,3)))
```

The sample output is as follows:

```
Random matrix of shape (4,3):
 [[6 7 6]
 [5 6 7]
 [5 3 6]
 [2 9 4]]
```

As we can see from the preceding output, a matrix was generated with a random shape.

Note

When creating matrices, you need to pass on tuples of integers as arguments. The output is susceptible to change since we have used random numbers.

Random number generation is a very useful utility and needs to be mastered for data science/data wrangling tasks. We will look at the topic of random variables and distributions again in the section on statistics and learn how NumPy and pandas have built-in random number and series generation, as well as manipulation functions.

Reshaping an array is a very useful operation for vectors as machine learning algorithms may demand input vectors in various formats for mathematical manipulation. In this section, we will be looking at how reshaping can be done on an array. The opposite of **`reshape`** is the **`ravel`** function, which flattens any given array into a one-dimensional array. It is a very useful action in many machine learning and data analytics tasks.

Exercise 9: Reshaping, Ravel, Min, Max, and Sorting

In this exercise, we will generate a random one-dimensional vector of two-digit numbers and then reshape the vector into multi-dimensional vectors. Let's go through the following steps:

1. Create an array of **`30`** random integers (sampled from **`1`** to **`99`**) and reshape it into two different forms using the following code:
   ```
   import numpy as np
   a = np.random.randint(1,100,30)
   b = a.reshape(2,3,5)
   ```

```
c = a.reshape(6,5)
```

2. Print the shape using the **`shape`** function by using the following code:

```
print ("Shape of a:", a.shape)
print ("Shape of b:", b.shape)
print ("Shape of c:", c.shape)
```

The output is as follows:

```
Shape of a: (30,)
Shape of b: (2, 3, 5)
Shape of c: (6, 5)
```

3. Print the arrays **`a`**, **`b`**, and **`c`** using the following code:

```
print("\na looks like\n",a)
print("\nb looks like\n",b)
print("\nc looks like\n",c)
```

The sample output is as follows:

```
a looks like
 [ 7 82 9 29 50 50 71 65 33 84 55 78 40 68 50
15 65 55 98
 38 23 75 50 57 32 69 34 59 98 48]
b looks like
 [[[ 7 82 9 29 50]
  [50 71 65 33 84]
  [55 78 40 68 50]]
 [[15 65 55 98 38]
  [23 75 50 57 32]
  [69 34 59 98 48]]]
c looks like
 [[ 7 82 9 29 50]
 [50 71 65 33 84]
 [55 78 40 68 50]
 [15 65 55 98 38]
 [23 75 50 57 32]
 [69 34 59 98 48]]
```

Note

b *is a three-dimensional array – a kind of list of a list of a list. The output is susceptible to change since we have used random numbers.*

4. Ravel file **b** using the following code:

```
b_flat = b.ravel()
print(b_flat)
```

The sample output is as follows (the output may be different in each iteration):

```
[ 7 82 9 29 50 50 71 65 33 84 55 78 40 68 50
15 65 55 98 38
 23 75 50 57 32 69 34 59 98 48]
```

In this exercise, you learned how to use **shape** and **reshape** functions to see and adjust the dimensions of an array. This can be useful in a variety of cases when working with arrays.

Indexing and slicing NumPy arrays is very similar to regular list indexing. We can even go through a vector of elements with a definite step size by providing it as an additional argument in the format (start, step, end). Furthermore, we can pass a list as an argument to select specific elements.

Note

In multi-dimensional arrays, you can use two numbers to denote the position of an element. For example, if the element is in the third row and second column, its indices are 2 and 1 (because of Python's zero-based indexing).

Exercise 10: Indexing and Slicing

In this exercise, we will learn how to perform indexing and slicing on one-dimensional and multi-dimensional arrays. To complete this exercise, let's go through the following steps:

1. Create an array of `10` elements and examine its various elements by slicing and indexing the array with slightly different syntaxes. Do this by using the following command:
   ```
   import numpy as np
   array_1 = np.arange(0,11)
   print("Array:",array_1)
   ```
   The output is as follows:
   ```
   Array: [ 0 1 2 3 4 5 6 7 8 9 10]
   ```
2. Print the element in the seventh position by using the following command:
   ```
   print("Element at 7th index is:", array_1[7])
   ```
   The output is as follows:
   ```
   Element at 7th index is: 7
   ```
3. Print the elements between the third and sixth positions by using the following command:
   ```
   print("Elements from 3rd to 5th index are:",
   array_1[3:6])
   ```
   The output is as follows:
   ```
   Elements from 3rd to 5th index are: [3 4 5]
   ```
4. Print the elements until the fourth position by using the following command:
   ```
   print("Elements up to 4th index are:",
   array_1[:4])
   ```
   The output is as follows:
   ```
   Elements up to 4th index are: [0 1 2 3]
   ```
5. Print the elements backward by using the following command:

```
print("Elements from last backwards are:",
array_1[-1::-1])
```
The output is as follows:
```
Elements from last backwards are: [10 9 8 7 6
5 4 3 2 1 0]
```
6. Print the elements using their backward index, skipping three values, by using the following command:
```
print("3 Elements from last backwards are:",
array_1[-1:-6:-2])
```
The output is as follows:
```
3 Elements from last backwards are: [10 8 6]
```
7. Create a new array called **`array_2`** by using the following command:
```
array_2 = np.arange(0,21,2)
print("New array:",array_2)
```
The output is as follows:
```
New array: [ 0 2 4 6 8 10 12 14 16 18 20]
```
8. Print the second, fourth, and ninth elements of the array:
```
print("Elements at 2nd, 4th, and 9th index
are:", \
      array_2[[2,4,9]])
```
The output is as follows:
```
Elements at 2nd, 4th, and 9th index are: [ 4
8 18]
```
9. Create a multi-dimensional array by using the following command:
```
matrix_1 =
np.random.randint(10,100,15).reshape(3,5)
print("Matrix of random 2-digit numbers\n
",matrix_1)
```
The sample output is as follows:
```
Matrix of random 2-digit numbers
```

```
 [[21 57 60 24 15]
 [53 20 44 72 68]
 [39 12 99 99 33]]
```

Note

The output is susceptible to change since we have used random numbers.

10. Access the values using double bracket indexing by using the following command:

```
print("\nDouble bracket indexing\n")
print("Element in row index 1 and column
index 2:", \
      matrix_1[1][2])
```

The sample output is as follows:

```
Double bracket indexing
Element in row index 1 and column index 2: 44
```

11. Access the values using single bracket indexing by using the following command:

```
print("\nSingle bracket with comma
indexing\n")
print("Element in row index 1 and column
index 2:", \
      matrix_1[1,2])
```

The sample output is as follows:

```
Single bracket with comma indexing
Element in row index 1 and column index 2: 44
```

12. Access the values in a multi-dimensional array using a row or column by using the following command:

```
print("\nRow or column extract\n")
print("Entire row at index 2:", matrix_1[2])
print("Entire column at index 3:",
matrix_1[:,3])
```

The sample output is as follows:

```
Row or column extract
Entire row at index 2: [39 12 99 99 33]
Entire column at index 3: [24 72 99]
```

13. Print the matrix with the specified row and column indices by using the following command:

```
print("\nSubsetting sub-matrices\n")
print("Matrix with row indices 1 and 2 and column "\
      "indices 3 and 4\n", matrix_1[1:3,3:5])
```

The sample output is as follows:

```
Subsetting sub-matrices
Matrix with row indices 1 and 2 and column indices 3 and 4
 [[72 68]
 [99 33]]
```

14. Print the matrix with the specified row and column indices by using the following command:

```
print("Matrix with row indices 0 and 1 and column "\
      "indices 1 and 3\n", matrix_1[0:2,[1,3]])
```

The sample output is as follows:

```
Matrix with row indices 0 and 1 and column indices 1 and 3
 [[57 24]
 [20 72]]
```

Note

The output is susceptible to change since we have used random numbers.

In this exercise, we worked with **`NumPy`** arrays and various ways of subletting them, such as slicing them. When working with arrays, it's very common to deal with them in this way.

Conditional SubSetting

Conditional subsetting is a way to select specific elements based on some numeric condition. It is almost like a shortened version of a SQL query to subset elements. See the following example:

```
matrix_1 =
np.array(np.random.randint(10,100,15)).reshape(3,5)
print("Matrix of random 2-digit
numbers\n",matrix_1)
print ("\nElements greater than 50\n",
matrix_1[matrix_1>50])
```

In the preceding code example, we have created an array with 15 random values between `10-100`. We have applied the `reshape` function. Then, we selected the elements that are less than `50`.

The sample output is as follows (note that the exact output will be different for you as it is random):

```
Matrix of random 2-digit numbers
 [[71 89 66 99 54]
 [28 17 66 35 85]
 [82 35 38 15 47]]
Elements greater than 50
 [71 89 66 99 54 66 85 82]
```

NumPy arrays operate just like mathematical matrices, and the operations are performed element-wise.

Now, let's look at an exercise to understand how we can perform array operations.

Exercise 11: Array Operations

In this exercise, we're going to create two matrices (multi-dimensional arrays) with random integers and demonstrate element-wise mathematical operations such as addition, subtraction, multiplication, and division. We can show the exponentiation (raising a number to a certain power) operation by performing the following steps:

Note

Due to random number generation, your specific output could be different than what is shown here.

1. Import the **`NumPy`** library and create two matrices:

   ```
   import numpy as np
   matrix_1 =
   np.random.randint(1,10,9).reshape(3,3)
   matrix_2 =
   np.random.randint(1,10,9).reshape(3,3)
   print("\n1st Matrix of random single-digit
   numbers\n",\
         matrix_1)
   print("\n2nd Matrix of random single-digit
   numbers\n",\
         matrix_2)
   ```

   The sample output is as follows (note that the exact output will be different for you as it is random):

   ```
   1st Matrix of random single-digit numbers
    [[6 5 9]
    [4 7 1]
    [3 2 7]]
   2nd Matrix of random single-digit numbers
    [[2 3 1]
    [9 9 9]
    [9 9 6]]
   ```

2. Perform addition, subtraction, division, and linear combination on the matrices:

```
print("\nAddition\n", matrix_1+matrix_2)
print("\nMultiplication\n",
matrix_1*matrix_2)
print("\nDivision\n", matrix_1/matrix_2)
print("\nLinear combination: 3*A - 2*B\n", \
      3*matrix_1-2*matrix_2)
```

The sample output is as follows (note that the exact output will be different for you as it is random):

```
Addition
 [[ 8 8 10]
 [13 16 10]
 [12 11 13]]
Multiplication
 [[12 15 9]
 [36 63 9]
 [27 18 42]]
Division
 [[3.       1.66666667 9.     ]
 [0.44444444 0.77777778 0.11111111]
 [0.33333333 0.22222222 1.16666667]]
Linear combination: 3*A - 2*B
 [[ 14    9 25]
 [ -6    3 -15]
 [ -9 -12    9]]
```

3. Perform the addition of a scalar, exponential matrix cube, and exponential square root:

```
print("\nAddition of a scalar (100)\n",
100+matrix_1)
print("\nExponentiation, matrix cubed
here\n", matrix_1**3)
print("\nExponentiation, square root using
'pow' function\n", \
      pow(matrix_1,0.5))
```

The sample output is as follows (note that the exact output will be different for you as it is random):

```
Addition of a scalar (100)
 [[106 105 109]
 [104 107 101]
 [103 102 107]]
Exponentiation, matrix cubed here
 [[216 125 729]
 [ 64 343   1]
 [ 27   8 343]]
Exponentiation, square root using 'pow'
function
 [[2.44948974 2.23606798 3.        ]
 [2.         2.64575131 1.        ]
 [1.73205081 1.41421356 2.64575131]]
```

Note

The output is susceptible to change since we have used random numbers.

We have now seen how to work with arrays to perform various mathematical functions, such as scalar addition and matrix cubing.

Stacking Arrays

Stacking arrays on top of each other (or side by side) is a useful operation for data wrangling. Stacking is a way to concatenate two NumPy arrays together. Here is the code:

```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
print("Matrix a\n",a)
print("Matrix b\n",b)
print("Vertical stacking\n",np.vstack((a,b)))
print("Horizontal stacking\n",np.hstack((a,b)))
```

The output is as follows:

```
Matrix a
 [[1 2]
 [3 4]]
Matrix b
 [[5 6]
 [7 8]]
Vertical stacking
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
Horizontal stacking
 [[1 2 5 6]
 [3 4 7 8]]
```

`NumPy` has many other advanced features, mainly related to statistics and linear algebra functions, which are used extensively in machine learning and data science tasks. However, not all of that is directly useful for beginner-level data wrangling, so we won't cover it here.

With all this knowledge (Numpy, Pandas, Matplotlib), let's try our hand at Exercise 12 (activity).

Exercise 12: Generating Statistics from a CSV File

Suppose you are working with the Boston Housing price dataset. This dataset is famous in the machine learning community. Many regression problems can be formulated, and machine learning algorithms can be run on this dataset. You will perform a basic data wrangling activity (including plotting some trends) on this dataset (**`.csv`** file) by reading it as a **`pandas`** DataFrame. We will perform a few statistical operations on this DataFrame.

Note

The Boston Housing dataset can be found here: **`Boston_housing.csv`**. *The* **`pandas`** *function for reading a CSV file is* **`read_csv`**.

These steps will help you complete this activity:

1. Load the necessary libraries.
2. Read in the Boston Housing dataset (given as a **`.csv`** file) from the local directory.
3. Check the first **`10`** records. Find the total number of records.
4. Create a smaller DataFrame with columns that do not include **`CHAS`**, **`NOX`**, **`B`**, and **`LSTAT`**:
   `Chas`: Charlse River Dummy variable
   `Nox`: Nitric Oxide concentration
   `B`: Proportion of the population that is African American
   `LSTAT`: Percentage of lower-income population
5. Check the last seven records of the new DataFrame you just created.
6. Plot the histograms of all the variables (columns) in the new DataFrame.
7. Plot them all at once using a for loop. Try to add a unique title to the plot.
8. Create a scatter plot of crime rate versus price.
9. Plot **`log10(crime)`** versus **`price`**.

10. Calculate some useful statistics, such as mean rooms per dwelling, median age, mean distances to five Boston employment centers, and the percentage of houses with a low price (`< $20,000`). **Hint:** To calculate the percentage of houses below `$20,000`, create a `pandas` series with the `PRICE` column and directly compare it with `20`. You can do this because `pandas` series is basically a `NumPy` array and you have seen how to filter NumPy array in the exercises in this chapter.

The output should be as follows:

Mean rooms per dwelling: `6.284634387351788`

Median age: `77.5`

Mean distances to five Boston employment centers: `3.795042687747034`

Percentage of houses with a low price (<$20,000): `41.50197628458498`