2
Getting Started with NumPy, pandas, and Matplotlib

In the previous chapter, we looked at some key financial topics. In this chapter, we will take a look at various data analysis libraries that we will use alongside Python.

As the financial domain deals with lots of data that may be in different formats and structures, we often have to analyze, preprocess, and visualize it in order to understand it efficiently. The NumPy, pandas, and matplotlib libraries are very important libraries that are used in Python for data analysis and data preprocessing.

The following topics will be covered in this chapter:

- NumPy
- pandas
- Matplotlib

So, let's get started.

Technical requirements

In this chapter, we will be using the Jupyter Notebook. You also need to install the NumPy, pandas, and matplotlib libraries if they are not already installed.

The code files for this chapter can be found at `https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%202`.

Another prerequisite of this chapter is that you have Anaconda installed on your computer.

A brief introduction to NumPy

In this section, we will briefly discuss NumPy, its uses, and how to install it on your computer. NumPy is a linear algebra library for Python. All other libraries in the PyData ecosystem rely on NumPy, as it is one of the fundamental building blocks.

Since NumPy has bindings to C libraries, it is incredibly fast. PyData is a result of conferences organized by NumFOCUS, a not-for-profit organization that supports open source software. They frequently hold gatherings in Silicon Valley, Boston, NYC, and, most recently, in London. A significant number of the meeting coordinators are based in Austin, TX, at the Continuum Analytics organization, which was established by Travis Oliphant and Peter Wang. Leah Silen is the fundamental coordinator of the gatherings, but there are also volunteer coordinators who help with organizing the conferences.

PyData refers to the group that is primarily involved in using Python and its various libraries for data preprocessing and data analysis. It is more business-centered than SciPy, which is made up of the company Enthought and is more focused on academic applications. The two networks have a lot in common, but you'll discover more finance-related themes with PyData.

To install NumPy, follow these steps:

1. It is highly recommended that you install Python using the Anaconda distribution to make sure that all the underlying dependencies (such as linear algebra libraries) sync up with conda installation.
2. If you already have Anaconda, install NumPy by going to your Terminal or Anaconda Command Prompt and typing the following command:

```
conda install numpy
```

3. If you do not have the Anaconda distribution and you have installed Python manually, type the following command from the Terminal:

```
pip install numpy
```

Now that we have installed NumPy, let's discuss the following topics:

- NumPy arrays
- NumPy indexing
- Various NumPy operations

NumPy arrays

NumPy is a library that is used in the Python programming language to create single and multidimensional arrays and matrices. It also supports various built-in functions that can perform high-level mathematical functions and operations on these arrays.

There are two main variants of NumPy arrays, as follows:

- Vectors
- Matrices

Vectors are arrays that are strictly one-dimensional, whereas matrices are two-dimensional. They can still only have, however, one row or column.

In this section, we are going to learn about the various ways to create NumPy arrays using Python and the NumPy library. We are going to use the Jupyter Notebook to show the programming code.

To begin using the NumPy library, we need to import it, as follows:

```
ln [1]: import numpy as np
```

Let's create a NumPy array using a Python object. First, we will create a list and then convert it to a NumPy array:

```
In [12]:
#Example of a list
list_1 = [1,2,3]
#show
list_1

Out[12]:
[1, 2, 3]
```

We can also assign this array to a variable and use it where necessary:

```
In [14]:
#Assign array to variable
list_array=np.array(list_1)
#Print list_array
print(list_array)

Out[14]
[1 2 3]
```

We can cast a normal Python list into an array. Currently, this is an example of a one-dimensional array. We can also create an array by directly converting a list of lists or a nested list:

```
In [15]:
nested_list= [[1,2,3],[4,5,6],[7,8,9]]
#show
nested_list

Out[15]:
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Let's now take a look at some of the built-in methods to generate arrays using NumPy.

NumPy's arange function

`arange` is a built-in function provided by the NumPy library to create an array with even-spaced elements according to a predefined interval. The syntax for NumPy's `arange` function is as follows:

```
arange([start,] stop[, step,], dtype=None)
```

The following examples demonstrate the basic usage of the `arange` function:

```
In [9]: np.arange(0,10,2)
Out[9]: array([ 0,  2,  4,  6,  8, 10])
```

NumPy's zeros and ones functions

The `zeros` and `ones` functions are built-in functions used to create arrays of zeros or ones; the syntax is as follows:

```
np.zeros(shape)
np.ones(shape)
```

The following examples show us how to create an array using `zeros` and `ones`:

```
In [24]:
np.zeros(3)

Out[24]:
array([ 0.,  0.,  0.])

In [26]:
np.zeros((5,5))
```

```
Out[26]:
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.]])

In [27]:
np.ones(3)

Out[27]:
array([ 1.,  1.,  1.])

In [28]:
np.ones((3,3))

Out[28]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

NumPy's linspace function

The `linspace` function is a built-in function that returns an evenly-spaced number over a specified interval. `linspace` requires three important attributes: `start`, `stop`, and `num`. By default, the `num` value is `50`, but this can be changed.

The syntax for NumPy's `linspace` function is as follows:

```
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
Docstring: Return evenly spaced numbers over a specified interval

In [6]:
np.linspace(0,10,3)

Out[6]:
array([ 0.,  5., 10.])

In [7]:
np.linspace(0,10,50)
```

The following output essentially returns 50 evenly-spaced numbers over a specified interval between 0 and 10:

```
Out[7]:
array([ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

NumPy's eye function

The `eye` function is a built-in function in NumPy that is used to create an identity matrix. An identity matrix is a type of matrix in which all the diagonal elements are one, and the remaining elements are zero:

```
In [8]:
np.eye(4)

Out[8]:
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

NumPy's rand function

NumPy has a built-in function called `rand` to create an array with random numbers. The `rand` function creates an array of a specified shape and populates it with random samples from a uniform distribution over [0, 1]. This essentially means that a random value between 0 and 1 will be selected, as shown in the following code block:

```
In [47]:
np.random.rand(2)

Out[47]:
array([ 0.11570539,  0.35279769])

In [46]:
```

```
np.random.rand(5,5)

Out[46]:

array([[ 0.66660768,  0.87589888,  0.12421056,  0.65074126,  0.60260888],
       [ 0.70027668,  0.85572434,  0.8464595 ,  0.2735416 ,  0.10955384],
       [ 0.0670566 ,  0.83267738,  0.9082729 ,  0.58249129,  0.12305748],
       [ 0.27948423,  0.66422017,  0.95639833,  0.34238788,  0.9578872 ],
       [ 0.72155386,  0.3035422 ,  0.85249683,  0.30414307,  0.79718816]])
```

The `randn` function creates an array of a specified shape and populates it with random samples from a standard normal distribution:

```
In [1]:
import numpy as np
In [2]:
np.random.randn(2)

Out[2]:
array([ 0.1426585 , -0.79882962])

In [3]:
np.random.randn(5,5)

Out[3]:
array([[-0.31525094, -0.76859012,  0.72035964,  0.7312833 , -0.57112783],
       [ 0.47523585,  0.18562321, -1.42741078, -0.50190548,  0.39230943],
       [-0.06597815, -0.92100907,  0.27146975, -0.84471005, -0.09242036],
       [-1.70155241, -0.79810538,  0.04569422,  0.1908103 ,  0.15467256],
       [ 0.36371628, -0.39255851,  0.02732152, -1.62381529,  0.42104139]])
```

The `randint` function creates an array by returning random integers, from a low value (inclusive) to a high value (exclusive). It is a built-in function of the `random` module in Python 3. The `random` module provides access to various useful functions, one of which is `randint()`.

The syntax of the `randint()` function is as follows:

```
randint(start, end)
```

The following are the parameters required inside the `randint()` function:

`(start, end)` : `Both of them must be integer type values.`

The following is the value returned from the `randint()` function:

```
A random integer within the given range as parameters.
```

The following are the errors and exceptions usually given by the `randint()` function based on the input:

```
ValueError : Returns a ValueError when floating
             point values are passed as parameters.

TypeError : Returns a TypeError when anything other than
```

```
In [6]:
np.random.randint(1,100)

Out[6]:
1

In [7]:
np.random.randint(1,100,10)

Out[7]:
array([95, 11, 47, 22, 63, 84, 16, 91, 67, 94])

Out[11]:
array([0, 2, 4, 6, 8])
```

Now let's discuss NumPy indexing. Here, indexing will help us to retrieve the elements of the array in a much faster way.

NumPy indexing

In this section, we are going to see how we can select elements or a group of elements from an array. We are also going to look at the indexing and selection mechanisms. First, let's create a simple one-dimensional array and look at how the indexing mechanism works:

```
In [2]:
import numpy as np

In [3]:
#Creating a sample array
arr_example = np.arange(0,11)

In [4]:
#Show the array
arr_example

Out[4]:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In the preceding code, we created an array with the `arr` variable name. Let's now see how we can select elements using indexing:

```
In [5]:
#Get a value at an index
arr_example[8]

Out[5]:
8

In [6]:
#Get values in a range
arr_example[1:5]

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

In [7]:
#Get values in a range
arr_example[0:5]

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

We can also update a range of array values by using broadcasting techniques:

```
In [8]:
#Setting a value with index range (Broadcasting)
arr_example[0:5]=100

#Show
arr_example

Out[8]:
array([100, 100, 100, 100, 100,   5,   6,   7,   8,   9,  10])
```

Let's take a look at how the indexing mechanism works for a two-dimensional array.

The general format for a two-dimensional array is either `array_2d[row][col]` or `array_2d[row,col]`. It is recommended to use comma notation for clarity.

The first step is to create a two-dimensional array and use the same indexing mechanism with slicing techniques to retrieve the elements from the array:

```
In [4]:
arr_2d = np.array([[5,10,15],[20,25,30],[35,40,45]])

#Show
arr_2d

Out[4]:
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])

In [5]:
#Indexing row
arr_2d[1]

Out[5]:
array([20, 25, 30])

In [6]:
# Format is arr_2d[row][col] or arr_2d[row,col]

# Getting individual element value
arr_2d[1][0]

Out[6]:
20

In [7]:
# Getting individual element value
arr_2d[1,0]

Out[7]:
20
```

The slicing technique helps us to retrieve elements from an array with respect to their indexes. We need to provide the indexes to retrieve the specific data from the array. Let's take a look at a few examples of slicing in the following code:

```
In [8]:
# 2D array slicing technique
arr_2d[:2,1:]

Out[8]:
array([[10, 15],
       [25, 30]])
```

```
In [9]:
arr_2d[2]

Out[9]:
array([35, 40, 45])

In [10]:
#Shape bottom row
arr_2d[2,:]

Out[10]:
array([35, 40, 45])
```

NumPy operations

We can easily perform arithmetic operations using a NumPy array. These operations may be array operations, array arithmetic, or scalar operations:

```
In [3]:
import numpy as np
arr_example = np.arange(0,10)

In [4]:
arr_example + arr_example

Out[4]:
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [5]:
arr_example * arr_example

Out[5]:
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

In [6]:
arr_example - arr_example

Out[6]:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

NumPy has many universal array functions that are essential mathematical techniques that anyone can use to perform arithmetic operations:

```
In [10]:
#Taking Square Roots
np.sqrt(arr_example)
```

```
Out[10]:
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

In [11]:
#Calculating exponential (e^)
np.exp(arr_example)

Out[11]:
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])

In [12]:
np.max(arr_example) #same as arr.max()

Out[12]:
9
```

A brief introduction to pandas

In the previous section, we discussed the NumPy library, its built-in functions, and its applications. We are now going to move on to discussing the pandas library. The pandas library is very powerful, and is one of the most important tools for data analysis and data preprocessing. It is an open source library that is built on top of NumPy. It provides many important features, such as fast data analysis, data cleaning, and data preparation. We provide data as input to our machine learning or deep learning models for training purposes. pandas has high performance and high productivity; it also has many built-in visualization features. One of the most important attributes of the pandas library is that it can work with data from a variety of data sources, such as **comma-separated value** (**CSV**) files, HTML, JSON, and Excel.

In order to use the pandas library, we will need to install it by going to the Anaconda prompt, the Command Prompt, or the Terminal, and typing in the following command:

```
conda install pandas
```

We can also use the `pip install` command to install the pandas library:

```
pip install pandas
```

The following topics will be covered in this section:

- Series
- DataFrames
- pandas operations
- pandas data input and output

Series

Series is the first datatype that we will be discussing in pandas. A series is a one-dimensional array with custom labels or indexes that have the ability to hold different types of data, such as integer, string, float, and Python objects.

A simple series in pandas can be created using the following constructor:

```
pandas.Series( data, index, dtype, copy)
```

The following table is a detailed explanation of the parameters used inside series:

Serial number	Parameters	Description
1	`data`	Data can be provided in the form of lists, *n*-dimensional arrays, or constants.
2	`index`	The index values must be provided as unique values. They should be of the same length as the `data` value provided in the first parameter. If no index is passed, a series usually takes up `numpy.arange(n)` as default, as discussed in the *NumPy* section of `Chapter 1`, *Python for Finance 101*.
3	`dtype`	`dtype` stands for datatype. The default value for the `dtype` parameter is `None`. If `dtype` is `None`, the datatype will be inferred.
4	`copy`	This allows you to copy data; the default value is `false`.

A series can be created by using a variety of inputs, such as the following:

- An array input
- A dictionary input
- A constant or scalar input

Let's consider how we can create series using the Python and pandas libraries. First, we will import the NumPy and pandas libraries:

```
In [1]:
import numpy as np
import pandas as pd
```

We can create series using various datatypes, such as lists, arrays, dictionaries, and constants:

```
In [2]:
labels_example = ['a','b','c']
list_example = [10,20,30]
dictionary_example = {'a':10,'b':20,'c':30}
arr_example = np.array([10,20,30])
```

Series can be also be created from a list, as shown in the following code block:

```
In [3]:
pd.Series(data=list_example)
```

The output is as follows:

```
Out[3]:
0    10
1    20
2    30
dtype: int64
```

In the following code, we are passing `list_example`, the indexes are set to `labels_example`, and we are converting them into series:

```
In [4]:
pd.Series(data=list_example,index=labels_example)
```

The output is as follows:

```
Out[4]:
a    10
b    20
c    30
dtype: int64
```

The following example shows how to create a series from an array:

```
In [5]:
pd.Series(arr_example)
```

The output is as follows:

```
Out[5]:
0    10
1    20
2    30
dtype: int32
```

In the following example, we have passed `array_example` as our data and `labels_example` as our index, and converted them into series:

```
In [6]:
pd.Series(arr_example,labels_example)
```

The output is as follows:

```
Out[6]:
a    10
b    20
c    30
dtype: int32
```

The following example shows how to create a series from a dictionary:

```
In [7]:
d.Series(dictionary_example)
```

The output is as follows:

```
Out[7]:
a    10
b    20
c    30
dtype: int64
```

pandas series can also hold various objects, including built-in Python functions such as `data`, as demonstrated in the following code:

```
In [8]:
pd.Series(data=labels_example)
```

The output is as follows:

```
Out[8]:
0    a
1    b
2    c
dtype: object
```

In the following code, we are adding built-in functions, such as `print` and `len`, and converting into series:

```
In [10]:
# Even functions (although unlikely that you will use this)
pd.Series([print,len])
```

The output is as follows:

```
Out[10]:
0    <built-in function print>
1      <built-in function len>
dtype: object
```

Now that we have seen how to create series using different datatypes, we should also learn how to retrieve values in the series using indexes. pandas makes use of index values, which allow fast retrieval of data; this is shown in the following example:

```
In [14]:
series1 = pd.Series([1,2,3,4],index = ['A','B','C','D'])
series1
```

The output is as follows:

```
Out[15]:
A    1
B    2
C    3
D    4
dtype: int64
```

The following code shows additional examples of how to create a series:

```
In [16]:
series2 = pd.Series([1,2,5,4],index = ['B','C','D','E'])
In [17]:
series2
```

The output is as follows:

```
Out[17]:
B    1
C    2
D    5
E    4
dtype: int64
```

The following code demonstrates how we can retrieve the series values by using indexing:

```
In [19]:
# Retrieving through index
series1[1]

Out[19]:
2
```

Different arithmetic operations can be performed on series with respect to values. In the following code, we will subtract the series:

```
In [23]:
series1-series2
```

The output is as follows:

```
Out[23]:
A    NaN
B    1.0
C    1.0-1.0
E    NaN
dtype: float64
```

The following code helps us to add two series – `series1` and `series2`:

```
In [24]:
series1 + series2
```

The output is as follows:

```
Out[24]:
A    NaN
B    3.0
C    5.0
D    9.0
E    NaN
dtype: float64
```

We can even multiply two or any number of series; the code is as follows:

```
In [25]:
series1 * series2
```

The output is as follows:

```
Out[25]:
A     NaN
B     2.0
C     6.0
D    20.0
E     NaN
dtype: float64
```

`NaN` values are displayed here because the indexes in the two series don't match. If the index present in `series1` is not present in the variables of `series2`, the value displays a `NaN` value for that particular index.

DataFrames

A DataFrame is the most popular tool that we will use in the pandas library. A DataFrame is essentially a data structure, usually with a two-dimensional shape, where the data is in a tabular format, consisting of rows and columns. DataFrames are the workhorse of pandas and are directly inspired by the R programming language. You can think of a DataFrame as a bunch of series objects put together, sharing the same index.

Here are the most important features of DataFrames:

- The data in the columns is of different datatypes.
- They are mutable and can be changed with respect to their size.
- A lot of complex arithmetic operations can be performed on the rows and columns.
- All the rows and columns can be accessed via a unique feature called an **axis**.

A pandas DataFrame can be created using the following constructor:

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows:

Serial number	Parameter and description
1	`data`: The `data` parameter can be provided in the form of lists, *n*-dimensional arrays, series, constants, or other DataFrames. This parameter essentially forms the values of the columns.

2	`index`: The `index` value is provided for the row labels. These labels act as an index for the complete rows, which are made of various columns provided by the `data` parameters.
3	`columns`: This parameter is to provide the column names or headings.
4	`dtype`: This is the datatype of each column.
5	`copy`: This parameter is used for copying data; the default value is `FALSE`.

Usually, `data` parameter values are given in the form of a two-dimensional array in the shape of *(n, m)*. The second parameter, index length, should be equal to the *n* value, while the column length should be equal to the *m* value.

A DataFrame can be created using various different types of input, such as lists, dictionaries, NumPy arrays, series, or other DataFrames. Let's consider some examples; the first step is to import the NumPy and pandas libraries, as we will need to create some arrays and DataFrames. The remaining tasks are quite simple; we need to add the elements provided in the syntax.

Let's import the library, as follows:

```
In [5]:
import pandas as pd
import numpy as np
```

We will also be importing the `randn` library, which will help us to create random numbers:

```
In [6]:
from numpy.random import randn
np.random.seed(55)
```

The following code helps us to create a DataFrame:

```
In [7]:
dataframe =
pd.DataFrame(randn(4,5),index=['P','Q','R','S'],columns=['A','B','C','D','E
'])
In [8]:
dataframe.head()
```

The output is as follows:

Out[8]:

	A	B	C	D	E
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

Selection and indexing in DataFrames

This section is about using the selection and indexing mechanism to retrieve elements from DataFrames. The selection and indexing mechanism works in pretty much the same way as we saw in NumPy, but there are some syntax changes. There will be some properties, such as `loc` and `iloc` in pandas, that are not present in NumPy. In the following code, we will retrieve the elements from the DataFrame:

```
In [9]:
dataframe['A']
```

The output is as follows:

```
Out[9]:
P   -1.623731
Q   -0.381086
R    1.656445
S    1.368799
Name: A, dtype: float64
```

We can also pass a list of column names, as follows:

```
In [10]:
# Passing a list of column names
dataframe[['A','B']]
```

The output is as follows:

Out[10]:

	A	B
P	-1.623731	-0.101784
Q	-0.381086	-0.002290
R	1.656445	-1.189009
S	1.368799	0.258169

The columns in the DataFrames are essentially a combination of series:

```
In [12]:
type(dataframe['A'])

Out[12]:
pandas.core.series.Series
```

A new column can be created using various arithmetic operations:

```
In [14]:
dataframe['F'] = dataframe['A'] + dataframe['B']
In [15]:
dataframe
```

The output is as follows:

Out[15]:

	A	B	C	D	E	F
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953	-1.725515
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100	-0.383376
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873	0.467436
S	1.368799	0.258169	0.702352	0.888382	0.722220	1.626967

Similarly, DataFrames have a built-in function to drop a column:

```
In [16]:
dataframe.drop('F',axis=1)
```

The output is as follows:

Out[16]:

	A	B	C	D	E
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

From the following code, you can see that even though we have dropped the `F` column, we still see the column being displayed:

```
In [17]:
dataframe
```

The output is as follows:

Out[17]:

	A	B	C	D	E	F
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953	-1.725515
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100	-0.383376
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873	0.467436
S	1.368799	0.258169	0.702352	0.888382	0.722220	1.626967

So, how is this possible? `drop` is a very risky operation, as it allows us to drop or delete a column. In order to prevent us from doing this by mistake, there is an extra parameter, `inplace`, which is usually used to ensure that the user really wants to do a `drop` operation; let's take a look at an example:

```
In [17]:
dataframe.drop('F',axis=1,inplace=True)

In [19]:
dataframe
```

The output is as follows:

Out[19]:

	A	B	C	D	E
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

When the `axis` value is `1`, the columns are usually dropped. Similarly, to drop a row with a particular index, the `axis` value is set to `0`:

```
In [24]:
dataframe.drop('P',axis=0,inplace=True)
dataframe
```

The output is as follows:

Out[24]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

There are more built-in properties available for DataFrames that are to do with selecting rows and columns:

- **Selecting rows in DataFrames**: The built-in properties used for selecting rows are `dataframe.loc` and `dataframe.iloc`. In the first property, we usually provide the index name, while in the latter, we provide the index number.

  The following syntax is used for selecting the rows of a DataFrame:

```
In [32]:
dataframe.loc['Q']
```

The output is as follows:

```
Out[32]:
A   -0.381086
B   -0.002290
C    0.341615
D    0.897572
E   -0.361100
Name: Q, dtype: float64
```

Alternatively, we can select a value based on its position, instead of its label:

```
In [30]:
dataframe.iloc[0]
```

The output is as follows:

```
Out[30]:
A   -0.381086
B   -0.002290
C    0.341615
D    0.897572
E   -0.361100
Name: Q, dtype: float64
```

Here, the output returns the complete row with respect to the column values.

- **Selecting a subset of rows and columns in a DataFrame**: The same built-in property, `dataframe.iloc`, is used to select subsets of rows and columns in a DataFrame. This selection requires the use of the slicing operation that we discussed in the *NumPy* section in the previous chapter; the code is as follows:

```
In [33]:
# Select all the rows and columns
dataframe.iloc[:]
```

The output is as follows:

Out[33]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

In the following code, we will skip the last column from the DataFrame:

```
In [38]:
#Skip the last column from the dataframe
dataframe.iloc[:,:-1]
```

The output is as follows:

Out[38]:

	A	B	C	D
Q	-0.381086	-0.002290	0.341615	0.897572
R	1.656445	-1.189009	1.666429	-2.003439
S	1.368799	0.258169	0.702352	0.888382

In the following code, we will select the subset of `Q` and `R` rows along with the `B` and `C` columns:

```
In [38]:
#Select subset Q and R rows along with B and C columns
dataframe.iloc[:2,1:3]
```

The output is as follows:

Out[40]:

	B	C
Q	-0.002290	0.341615
R	-1.189009	1.666429

In the following code, we will select the subset of rows and columns using `dataframe.loc`:

```
In [41]:
# Select subset of rows and column using dataframe.loc
dataframe.loc['Q','A']

Out[41]:
-0.3810863829200849

In [42]:
dataframe
dataframe.loc[['Q','R'],['B','C']]
```

The output is as follows:

Out[42]:

	B	C
Q	-0.002290	0.341615
R	-1.189009	1.666429

DataFrames also allow conditional selection, if we provide the conditions in brackets:

```
In [43]:
dataframe
```

The output is as follows:

Out[43]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

The following code returns `True` if greater than zero:

```
In [43]:
dataframe>0
```

The output is as follows:

Out[44]:

	A	B	C	D	E
Q	False	False	True	True	False
R	True	False	True	False	False
S	True	True	True	True	True

The following code will return values that are greater than 1:

```
In [46]:
#return the values where the value> 0 else returns NaN
dataframe[dataframe>0]
```

The output is as follows:

Out[46]:

	A	B	C	D	E
Q	NaN	NaN	0.341615	0.897572	NaN
R	1.656445	NaN	1.666429	NaN	NaN
S	1.368799	0.258169	0.702352	0.888382	0.72222

The following code returns the complete rows and columns where the value of the `A` column is greater than `0`:

```
In [47]:
#Returns the complete rows and columns where the value of columns A > 0
dataframe[dataframe['A']>0]
```

The output is as follows:

Out[47]:

	A	B	C	D	E
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

If we need to apply multiple conditions within the DataFrame, we use `|` and `&` with parentheses:

```
In [51]:
dataframe[(dataframe['A']>0) & (dataframe['B'] > 0)]
```

The output is as follows:

Out[51]:

	A	B	C	D	E
S	1.368799	0.258169	0.702352	0.888382	0.72222

There is also a built-in function that can reset the index in the DataFrame to the default index, as in NumPy (such as *0*, *1*, *2*, *3*, ..., *n*):

```
In [52]:
dataframe
```

The output is as follows:

Out[52]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

The `reset_index()` method reset the index to default index as shown here:

```
In [53]:
# reset_index() reset to default 0,1...n index
dataframe.reset_index()
```

The output is as follows:

Out[53]:

	index	A	B	C	D	E
0	Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
1	R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
2	S	1.368799	0.258169	0.702352	0.888382	0.722220

Different operations in pandas DataFrames

There are many different operations and built-in functions available in pandas DataFrames. These are very useful for data analysis and data preprocessing. We can create a DataFrame and apply various built-in functions to make different tasks easier. Let's now create a DataFrame and try different operations on it:

```
In [4]:
#Creating a dataframe with dictionary items within it as key value pairs.
The keys are the column name
import pandas as pd
df =
pd.DataFrame({'col1':[10,11,12,13],'col2':[100,200,300,400],'col3':['abc','
def','ghi','xyz']})
df.head()
```

The output for the preceding code will be as follows:

Out[4]:

	col1	col2	col3
0	10	100	abc
1	11	200	def
2	12	300	ghi
3	13	400	xyz

Let's take a closer look at some of the built-in functions:

- `unique()`: The `unique` function returns unique values from a column or a row in the form of an array.
- `nunique()`: The `nunique` function provides the number of unique elements in a row or column.
- `value_counts()`: The `value_counts` function provides the number of elements present in a row or column.

Let's now take a look at the following code:

```
In [5]:
df['col2'].unique()

Out[5]:
array([100, 200, 300, 400], dtype=int64)

In [6]:
df['col2'].nunique()

Out[6]:
4

In [7]:
df['col2'].value_counts()
```

The output is as follows:

```
Out[7]:
100    1
400    1
300    1
200    1
Name: col2, dtype: int64
```

A DataFrame also provides an option to apply an operation within a function to all the elements in the DataFrame. Let's define a function and see how we can apply it to a DataFrame:

```
In [20]:
def squareofanumber(value):
    return value**2

In [21]:
df['col1']

Out[21]:
0    10
1    11
2    12
3    13
Name: col1, dtype: int64

In [22]:
#apply() function applies the function definition in all the elements
from a column
df['col1'].apply(squareofanumber)

Out[22]:
0    100
1    121
2    144
3    169
Name: col1, dtype: int64
```

The `apply()` function applies the function to all of the elements in the DataFrame.

Make sure the function definition is compatible with the values of the DataFrame with respect to the datatypes used, or it may produce a runtime error.

Data input and output operations with pandas

The pandas library provides a lot of built-in functions to read data from different data sources, such as CSV, Excel, JSON, and HTML. These features have made pandas the favorite library of many data scientists and machine learning developers.

Reading CSV files

pandas provides a built-in function, `read_csv()`, to read data from a CSV data source. The output returned from this function is essentially a DataFrame. First, we need to import the pandas library, as follows:

```
In [15]:
import pandas as pd
```

The function used to read the CSV file is `read_csv()`:

```
In [16]:
dataframe = pd.read_csv('Customers.csv')

In [17]:
dataframe.head()
```

The output is as follows:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

We can convert this DataFrame back into the CSV file using the `to_csv()` function:

```
In [18]:
type(dataframe)

Out[18]:
pandas.core.frame.DataFrame

In [19]:
dataframe.to_csv('customer_copy.csv',index=False)
```

The CSV file is saved in the same location as that of the current file. It can be downloaded from the GitHub repository here: `https://goo.gl/DJxn9x`.

Reading Excel files

pandas also has a built-in `read_excel()` function, which is used to read data from Excel files. The output that is returned is also a DataFrame. The parameter that is usually provided in the `read_excel()` function is the name of the Excel file and the sheet name:

```
In [3]:
import pandas as pd

In [9]:
df=pd.read_excel('Sample.xlsx',sheet_name='Sample')
df.head()
```

The output for the preceding code is as follows:

Out[9]:

	Row ID	Order Priority	Discount	Unit Price	Shipping Cost	Customer ID	Customer Name	Ship Mode	Customer Segment	Product Category	...	Region	State or Province
0	20847	High	0.01	2.84	0.93	3	Bonnie Potter	Express Air	Corporate	Office Supplies	...	West	Washingtor
1	20228	Not Specified	0.02	500.98	26.00	5	Ronnie Proctor	Delivery Truck	Home Office	Furniture	...	West	California
2	21776	Critical	0.06	9.48	7.29	11	Marcus Dunlap	Regular Air	Home Office	Furniture	...	East	New Jersey
3	24844	Medium	0.09	78.69	19.99	14	Gwendolyn F Tyson	Regular Air	Small Business	Furniture	...	Central	Minnesota
4	24846	Medium	0.08	3.28	2.31	14	Gwendolyn F Tyson	Regular Air	Small Business	Office Supplies	...	Central	Minnesota

5 rows × 25 columns

The `to_excel()` function is used to convert the DataFrame back into the Excel file. It is stored in the same location that we are currently working in:

```
In [10]:
df.to_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
```

The sample Excel file can be found at the GitHub link mentioned in the *Technical requirements* section.

Reading from HTML files

pandas also provides a built-in function to read data from HTML files. This function reads the table tabs from the HTML and returns a DataFrame object; an example of this is as follows:

```
In [12]:
df =
pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')

In [14]:
df[0].head()
```

The output for the preceding code is as follows:

Out[14]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date
0	Washington Federal Bank for Savings	Chicago	IL	30570	Royal Savings Bank	December 15, 2017	February 21, 2018
1	The Farmers and Merchants State Bank of Argonia	Argonia	KS	17719	Conway Bank	October 13, 2017	February 21, 2018
2	Fayette County Bank	Saint Elmo	IL	1802	United Fidelity Bank, fsb	May 26, 2017	July 26, 2017
3	Guaranty Bank, (d/b/a BestBank in Georgia & Mi...	Milwaukee	WI	30003	First-Citizens Bank & Trust Company	May 5, 2017	March 22, 2018
4	First NBC Bank	New Orleans	LA	58302	Whitney Bank	April 28, 2017	December 5, 2017

In this example, the `read_html()` function is used to read from the HTML URL. It returns a DataFrame object.

Matplotlib

Matplotlib is a very advanced data visualization library that is used with Python. It was created by John Hunter; he created it to try to replicate the plotting capabilities of MATLAB (another programming language) in Python. If you happen to be familiar with MATLAB, matplotlib will feel natural to you. It is an excellent two-dimensional and three-dimensional graphics library for generating scientific figures.

The major advantages of matplotlib include the following:

- It is generally very easy to get started for simple and complex plots.
- It provides features that support custom labels and texts.
- It provides great control of each and every element in a plotted figure.
- It provides high-quality output with many image formats.
- It provides lots of customizable options.

Matplotlib allows you to create reproducible figures programmatically. So, let's learn how to use it! Before continuing with this chapter, I encourage you to explore the official matplotlib web page here: `http://matplotlib.org/`.

In order to use the matplotlib library, we will need to install matplotlib by going to the Anaconda prompt, the Command Prompt, or the Terminal and typing in the following command:

```
conda install matplotlib
```

Alternatively, we can use the following command:

```
pip install matplotlib
```

Let's now move on and take a look at some examples of how to use matplotlib to visualize data. The first step is to import the library, as follows:

```
In [1]:
import matplotlib.pyplot as plt
```

You'll also need to use the following line to see plots in the Notebook:

```
In [2]:
%matplotlib inline
```

We are going to use some NumPy examples to create some arrays and learn how to plot them using the matplotlib library. Let's create an array using the `numpy.linspace` function, as follows:

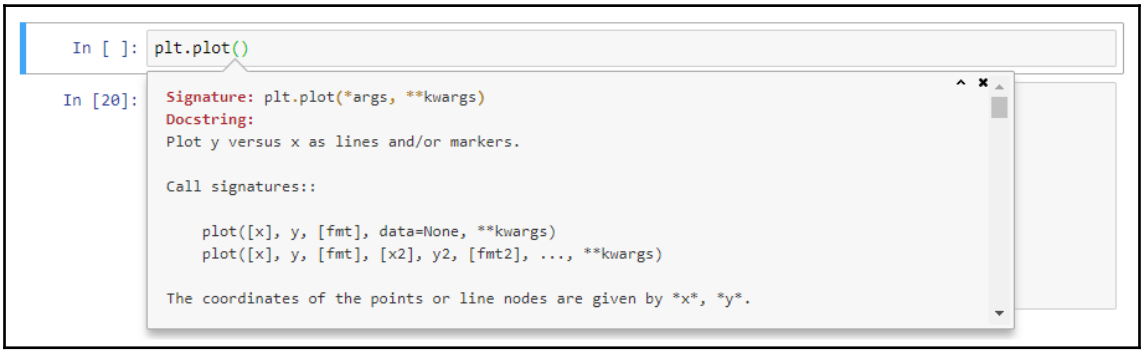
```
In [6]:
import numpy as np
import numpy as np
arr1= np.linspace(0, 10, 11)
arr2 = arr1 ** 2

In [7]:
arr1
```

```
Out[7]:
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])

In [8]:
arr2

Out[8]:
array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.,  64.,  81., 100.])
```

Here, two arrays, `arr1` and `arr2`, have been created using NumPy. Let's now take a look at some matplotlib commands to create a visualization graph.

The plot function

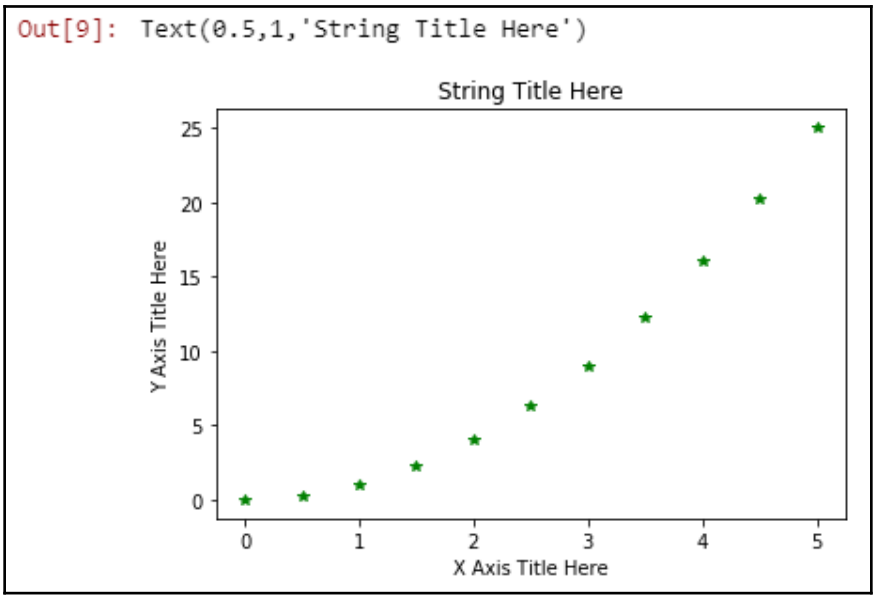
The `plot` function is used for plotting *y* versus *x* in the form of points, markers, or lines. The syntax of the `plot` function is as follows:



Usually, the first and second parameters are the arrays that we need to plot. There is a list of parameters that can be given to the `plot` function, such as the line color and the pixel size of the line. We can create a very simple line plot using the following code:

```
In [9]:
# 'g' is the color green
plt.plot(a,b, 'g*',label="Example 2")
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
```

The output is as follows:



The third parameter inside the `plot` function is essentially the color parameter, which is used to provide a color for the plotted points.

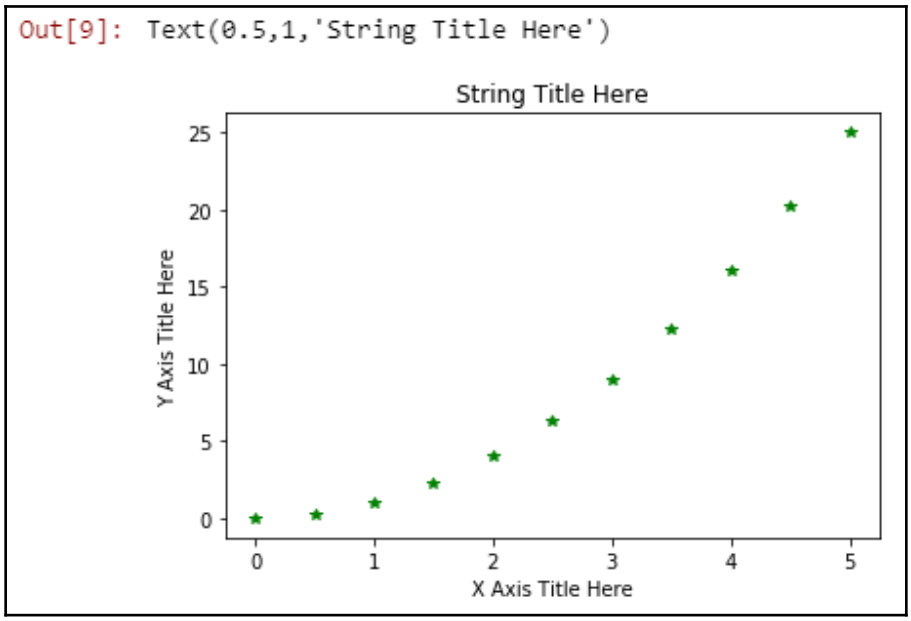
To find out more about the different parameters and features inside the various built-in functions provided by matplotlib, I encourage you to explore the official matplotlib web page here: `http://matplotlib.org/`.

The xlabel function

In the preceding example, we used the `xlabel` built-in function to provide a label to the *x* axis. This helps you to provide your own custom label values for the *x* axis. We can create a very simple line plot using the following code:

```
In [9]:
# 'g' is the color green
plt.plot(a,b, 'g*',label="Example 2")
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
```

The output is shown here:



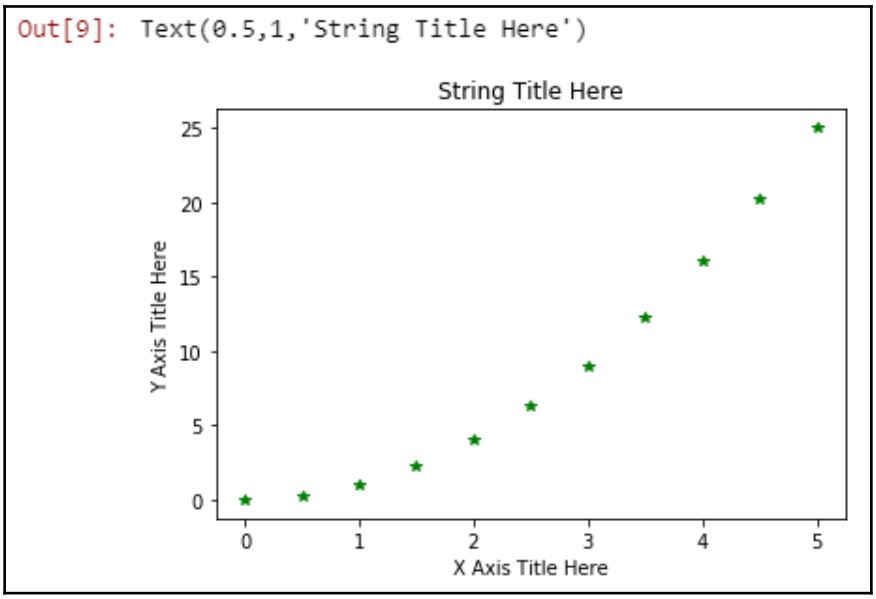
In the preceding example, we provided the `xlabel` value as `X Axis Title Here`.

The ylabel function

Similarly, there is another built-in function, `plt.ylabel()`, which we can use to provide a custom label for the *y* axis. We can create a very simple line plot using the following code:

```
In [9]:
# 'g' is the color green
plt.plot(a,b, 'g*',label="Example 2")
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
```

The output is shown here:



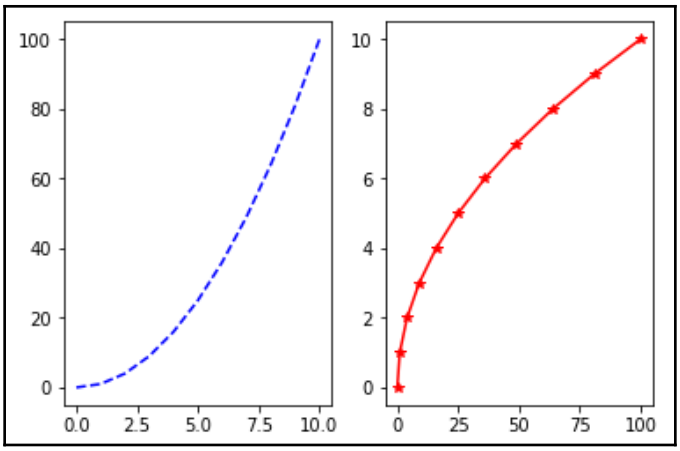
In the preceding example, we provided the value of `ylabel` as `Y Axis Title Here`.

Creating multiple subplots using matplotlib

We can also create multiple subplots using the `subplot` function, which is provided in the matplotlib library; let's take a look at an example:

```
In [14]:
import numpy as np
a= np.linspace(0, 10, 11)
b = a**2

In [15]:
# plt.subplot(nrows, ncols, plot_number)
plt.subplot(1,2,1)
plt.plot(a, b, 'b--') # More on color options later
plt.subplot(1,2,2)
plt.plot(b, a, 'r*-');
```

The output is as follows:



The `matplotlib` library also provides various advanced methods, such as `histogram` and `pie`, which can provide better visualization of data.

The pie function

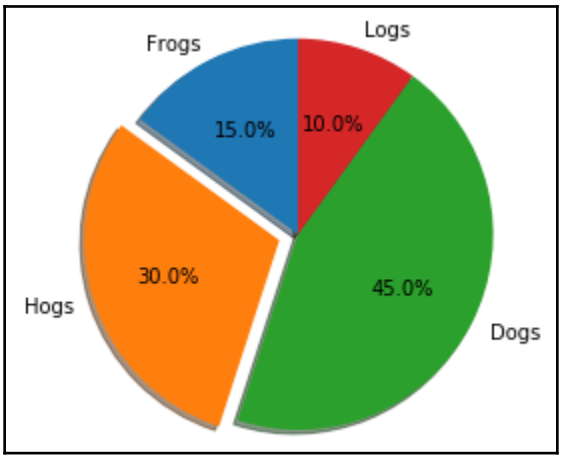
The `pie` function helps to categorize a group of entities so that they can be represented visually. In short, a `pie` function helps to plot a pie chart; the syntax is as follows:

```
matplotlib.pyplot.pie(x, explode=None, labels=None, colors=None,
autopct=None, pctdistance=0.6, shadow=False, labeldistance=1.1,
startangle=None, radius=None, counterclock=True, wedgeprops=None,
textprops=None, center=(0, 0), frame=False, rotatelabels=False, hold=None,
data=None)
```

Let's take a look at an example through the following code:

```
In [16]:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a
circle.

plt.show()
```

The output is shown here:



The histogram function

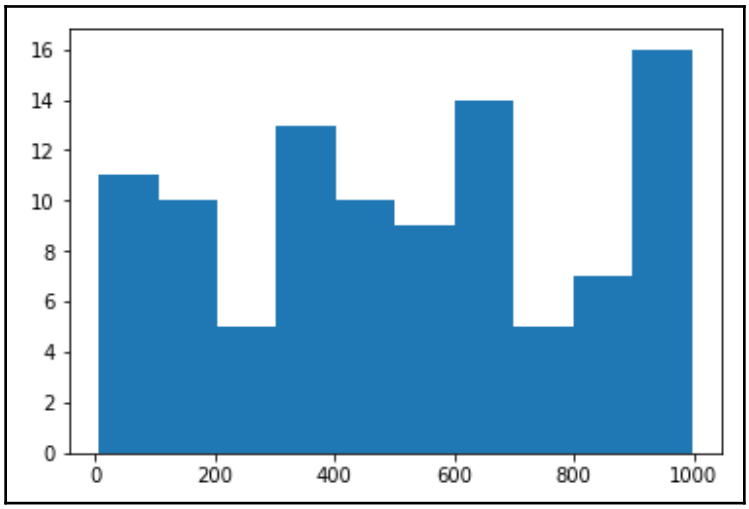
The `histogram` function in matplotlib plots histograms. The return value is a tuple, such as *n*, *bins*, and *patches*, or [*n0*, *n1*, ...], *bins*, and [*patches[0]*, *patches[1]*,...], if the input contains multiple data values; let's take a look at the following example:

```
In [18]:
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```

The output is as follows:

```
Out[18]:
(array([11., 10.,  5., 13., 10.,  9., 14.,  5.,  7., 16.]),
 array([  7. , 105.9, 204.8, 303.7, 402.6, 501.5, 600.4, 699.3, 798.2,
        897.1, 996. ]),
 <a list of 10 Patch objects>)
```

The following graph demonstrates the output generated:



Please refer to the main documentation website (`https://matplotlib.org`) for more information about matplotlib.

Summary

In this chapter, we discussed the NumPy, pandas, and matplotlib libraries. We used the NumPy library to create multidimensional arrays. It provides various functions that we can apply to arrays to perform complex arithmetic tasks. We also learned how to retrieve elements from an array using indexing techniques for both single-dimensional and multidimensional arrays.

Then, we discussed the pandas library. In the pandas library, we looked at series and DataFrames, which provide various built-in functions to read data from data sources such as CSV files, Excel files, and HTML files. The pandas library provides us with a convenient way to carry out data preprocessing, which is the backbone of data analysis.

Finally, we explored the matplotlib library. Here, we found an efficient way of visualizing data, which is also an important part of data analysis and data preprocessing. Visualization helps us to understand our data more efficiently and to make better decisions.

All these libraries help us to carry out efficient data analysis, data preprocessing, and data visualization tasks, which represent the core of any problem statements that we solve using Python. In the next chapter, we will learn how to use these libraries to solve some real-world problems.

Further reading

It is always a good idea to read the official documentation of the libraries discussed to gain additional knowledge.

You can refer to the following links for the documentation of each library:

- **NumPy**: `https://docs.scipy.org/doc/`
- **pandas**: `https://pandas.pydata.org/pandas-docs/stable/`
- **Matplotlib**: `https://matplotlib.org`