Chapter 10

NumPy Module

NumPy, an acronym for *Numerical Python*, is a package to perform scientific computing in Python efficiently. It includes random number generation capabilities, functions for basic linear algebra, Fourier transforms as well as a tool for integrating Fortran and C/C++ code along with a bunch of other functionalities.

NumPy is an open-source project and a successor to two earlier scientific Python libraries: *Numeric* and *Numarray*.

It can be used as an efficient multi-dimensional container of generic data. This allows NumPy to integrate with a wide variety of databases seamlessly. It also features a collection of routines for processing single and multidimensional vectors known as arrays in programming parlance.

NumPy is not a part of the Python Standard Library and hence, as with any other such library or module, it needs to be installed on a workstation before it can be used. Based on the Python distribution one uses, it can be installed via a command prompt, conda prompt, or terminal using the following command. *One point to note is that if we use the Anaconda distribution to install Python, most of the libraries (like NumPy, pandas, scikit-learn, matplotlib, etc.) used in the scientific Python ecosystem come pre-installed.*

```
pip install numpy
```

NOTE: If we use the Python or iPython console to install the

NumPy library, the command to install it would be preceded by the character `!`.

Once installed we can use it by importing into our program by using the `import` statement. The de facto way of importing is shown below:

```
import numpy as np
```

Here, the NumPy library is imported with an alias of `np` so that any functionality within it can be used with convenience. We will be using this form of alias for all examples in this section.

10.1 NumPy Arrays

A Python list is a pretty powerful sequential data structure with some nifty features. For example, it can hold elements of various data types which can be added, changed or removed as required. Also, it allows index subsetting and traversal. But lists lack an important feature that is needed while performing data analysis tasks. We often want to carry out operations over an entire collection of elements, and we expect Python to perform this fast. With lists executing such operations over all elements efficiently is a problem. For example, let's consider a case where we calculate PCR (Put Call Ratio) for the previous 5 days. Say, we have put and call options volume (in Lacs) stored in lists `call_vol` and `put_vol` respectively. We then compute the PCR by dividing put volume by call volume as illustrated in the below script:

```
# Put volume in lacs
In []: put_vol = [52.89, 45.14, 63.84, 77.1, 74.6]

# Call volume in lacs
In []: call_vol = [49.51, 50.45, 59.11, 80.49, 65.11]

# Computing Put Call Ratio (PCR)
In []: put_vol / call_vol
Traceback (most recent call last):

File "<ipython-input-12>", line 1, in <module>
```

```
    put_vol / call_vol

TypeError: unsupported operand type(s) for /: 'list' and
'list'
```

Unfortunately, Python threw an error while calculating PCR values as it has no idea on how to do calculations on lists. We can do this by iterating over each item in lists and calculating the PCR for each day separately. However, doing so is inefficient and tiresome too. A way more elegant solution is to use NumPy arrays, an alternative to the regular Python list.

The NumPy array is pretty similar to the list, but has one useful feature: we can perform operations over entire arrays(all elements in arrays). It's easy as well as super fast. Let us start by creating a NumPy array. To do this, we use `array()` function from the NumPy package and create the NumPy version of `put_vol` and `call_vol` lists.

```
# Importing NumPy library
In []: import numpy as np

# Creating arrays
In []: n_put_vol = np.array(put_vol)

In []: n_call_vol = np.array(call_vol)

In []: n_put_vol
Out[]: array([52.89, 45.14, 63.84, 77.1 , 74.6 ])

In []: n_call_vol
Out[]: array([49.51, 50.45, 59.11, 80.49, 65.11])
```

Here, we have two arrays `n_put_vol` and `n_call_vol` which holds put and call volume respectively. Now, we can calculate PCR in one line:

```
# Computing Put Call Ratio (PCR)
In []: pcr = n_put_vol / n_call_vol

In []: pcr
Out[]: array([1.06826904, 0.89474727, 1.0800203,
              0.95788297, 1.14575334])
```

This time it worked, and calculations were performed element-wise. The first observation in `pcr` array was calculated by dividing the first element in `n_put_vol` by the first element in `n_call_vol` array. The second element in `pcr` was computed using the second element in the respective arrays and so on.

First, when we tried to compute PCR with regular lists, we got an error, because Python cannot do calculations with lists like we want it to. Then we converted these regular lists to NumPy arrays and the same operation worked without any problem. NumPy work with arrays as if they are scalars. But we need to pay attention here. NumPy can do this easily because it assumes that array can only contain values of a single type. It's either an array of integers, floats or booleans and so on. If we try to create an array of different types like the one mentioned below, the resulting NumPy array will contain a single type only. String in the below case:

```
In []: np.array([1, 'Python', True])
Out[]: array(['1', 'Python', 'True'], dtype='<U11')
```

NOTE: NumPy arrays are made to be created as homogeneous arrays, considering the mathematical operations that can be performed on them. It would not be possible with heterogeneous data sets.

In the example given above, an integer and a boolean were both converted to strings. NumPy array is a new type of data structure type like the Python list type that we have seen before. This also means that it comes with its own methods, which will behave differently from other types. Let us implement the + operation on the Python list and NumPy arrays and see how they differ.

```
# Creating lists
In []: list_1 = [1, 2, 3]

In []: list_2 = [5, 6, 4]

# Adding two lists
In []: list_1 + list_2
Out[]: [1, 2, 3, 5, 6, 4]
```

```
# Creating arrays
In []: arr_1 = np.array([1, 2, 3])

In []: arr_2 = np.array([5, 6, 4])

# Adding two arrays
In []: arr_1 + arr_2
Out[]: array([6, 8, 7])
```

As can be seen in the above example, performing the + operation with `list_1` and `list_2`, the list elements are pasted together, generating a list with 6 elements. On the other hand, if we do this with NumPy arrays, Python will do an element-wise sum of the arrays.

10.1.1 N-dimensional arrays

Until now we have worked with two arrays: `n_put_vol` and `n_call_vol`. If we are to check its type using `type()`, Python tells us that they are of type `numpy.ndarray` as shown below:

```
# Checking array type
In []: type(n_put_vol)
Out[]: numpy.ndarray
```

Based on the output we got, it can be inferred that they are of data type `ndarray` which stands for *n-dimensional array* within NumPy. These arrays are one-dimensional arrays, but NumPy also allows us to create two dimensional, three dimensional and so on. We will stick to two dimensional for our learning purpose in this module. We can create a 2D (two dimensional) NumPy array from a regular Python list of lists. Let us create one array for all put and call volumes.

```
# Recalling put and call volumes lists
In []: put_vol
Out[]: [52.89, 45.14, 63.84, 77.1, 74.6]

In []: call_vol
Out[]: [49.51, 50.45, 59.11, 80.49, 65.11]
```

```
# Creating a two-dimensional array
In []: n_2d = np.array([put_vol, call_vol])

In []: n_2d
Out[]:
array([[52.89, 45.14, 63.84, 77.1 , 74.6 ],
       [49.51, 50.45, 59.11, 80.49, 65.11]])
```

We see that `n_2d` array is a rectangular data structure. Each list provided in the `np.array` creation function corresponds to a row in the two-dimensional NumPy array. Also for 2D arrays, the NumPy rule applies: an array can only contain a single type. If we change one float value in the above array definition, all the array elements will be coerced to strings, to end up with a homogeneous array. We can think of a 2D array as an advanced version of lists of a list. We can perform element-wise operation with 2D as we had seen for a single dimensional array.

10.2 Array creation using built-in functions

An explicit input has been provided while creating `n_call_vol` and `n_put_vol` arrays. In contrast, NumPy provides various built-in functions to create arrays and input to them will be produced by NumPy. Below we discuss a handful of such functions:

- `zeros(shape, dtype=float)` returns an array of a given shape and type, filled with zeros. If the *dtype* is not provided as an input, the default type for the array would be float.

```
# Creating a one-dimensional array
In []: np.zeros(5)
Out[]: array([0., 0., 0., 0., 0.])

# Creating a two-dimensional array
In []: np.zeros((3, 5))
Out[]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
```

```
       [0., 0., 0., 0., 0.]])

# Creating a one-dimensional array of integer type
In []: np.zeros(5, dtype=int)
Out[]: array([0, 0, 0, 0, 0])
```

- `ones(shape, dtype=float)` returns an array of a given shape and type, filled with ones. If the *dtype* is not provided as an input, the default type for the array would be float.

```
# Creating a one-dimensional array
In []: np.ones(5)
Out[]: array([1., 1., 1., 1., 1.])

# Creating a one-dimensional array of integer type
In []: np.ones(5, dtype=int)
Out[]: array([1, 1, 1, 1, 1])
```

- `full(shape, fill_value, dtype=None)` returns an array of a given shape and type, fill with *fill_value* given in input parameters.

```
# Creating a one-dimensional array with value as 12
In []: np.full(5, 12)
Out[]: array([12, 12, 12, 12, 12])

# Creating a two-dimensional array with value as 9
In []: np.full((2, 3), 9)
Out[]:
array([[9, 9, 9],
       [9, 9, 9]])
```

- `arange([start, ]stop, [step])` returns an array with evenly spaced values within a given interval. Here the *start* and *step* parameters are optional. If they are provided NumPy will consider them while computing the output. Otherwise, range computation starts from 0. For all cases, stop value will be excluded in the output.

```
# Creating an array with only stop argument
In []: np.arange(5)
```

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

# Creating an array with start and stop arguments
In []: np.arange(3, 8)
Out[]: array([3, 4, 5, 6, 7])

# Creating an array with given interval and step value
# as 0.5
In []: np.arange(3, 8, 0.5)
Out[]: array([3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5, 7. ,
              7.5])
```

- `linspace(start, stop, num=50, endpoint=True)` returns evenly spaced numbers over a specified interval. The number of samples to be returned is specified by the *num* parameter. The endpoint of the interval can optionally be excluded.

```
# Creating an evenly spaced array with five numbers within
# interval 2 to 3
In []: np.linspace(2.0, 3.0, num=5)
Out[]: array([2.  , 2.25, 2.5 , 2.75, 3.  ])

# Creating an array excluding end value
In []: np.linspace(2.0, 3.0, num=5, endpoint=False)
Out[]: array([2. , 2.2, 2.4, 2.6, 2.8])

# Creating an array with ten values within the specified
# interval
In []: np.linspace(11, 20, num=10)
Out[]: array([11., 12., 13., 14., 15., 16., 17., 18., 19.,
              20.])
```

10.3 Random Sampling in NumPy

In addition to built-in functions discussed above, we have a `random` sub-module within the NumPy that provides handy functions to generate data randomly and draw samples from various distributions. Some of the widely used such functions are discussed here.

- `rand([d0, d1, ..., dn])` is used to create an array of a given shape and populate it with random samples from a *uniform distribution* over `[0, 1)`. It takes only positive arguments. If no argument is provided, a single float value is returned.

```
# Generating single random number
In []: np.random.rand()
Out[]: 0.1380210268817208

# Generating a one-dimensional array with four random
# values
In []: np.random.rand(4)
Out[]: array([0.24694323, 0.83698849, 0.0578015,
              0.42668907])

# Generating a two-dimensional array
In []: np.random.rand(2, 3)
Out[]:
array([[0.79364317, 0.15883039, 0.75798628],
       [0.82658529, 0.12216677, 0.78431111]])
```

- `randn([d0, d1, ..., dn])` is used to create an array of the given shape and populate it with random samples from a *standard normal* distributions. It takes only positive arguments and generates an array of shape `(d0, d1, ..., dn)` filled with random floats sampled from a univariate normal distribution of mean 0 and variance 1. If no argument is provided, a single float randomly sampled from the distribution is returned.

```
# Generating a random sample
In []: np.random.randn()
Out[]: 0.5569441449249491

# Generating a two-dimensional array over N(0, 1)
In []: np.random.randn(2, 3)
Out[]:
array([[ 0.43363995, -1.04734652, -0.29569917],
       [ 0.31077962, -0.49519421,  0.29426536]])
```

```
# Generating a two-dimensional array over N(3, 2.25)
In []: 1.5 * np.random.randn(2, 3) + 3
Out[]:
array([[1.75071139, 2.81267831, 1.08075029],
       [3.35670489, 3.96981281, 1.7714606 ]])
```

- `randint(low, high=None, size=None)` returns a random integer from a discrete uniform distribution with limits of *low* (inclusive) and *high* (exclusive). If *high* is None (the default), then results are from 0 to *low*. If the *size* is specified, it returns an array of the specified size.

```
# Generating a random integer between 0 and 6
In []: np.random.randint(6)
Out[]: 2

# Generating a random integer between 6 and 9
In []: np.random.randint(6, 9)
Out[]: 7

# Generating a one-dimensional array with values between 3
# and 9
In []: np.random.randint(3, 9, size=5)
Out[]: array([6, 7, 8, 8, 5])

# Generating a two-dimensional array with values between 3
# and 9
In []: np.random.randint(3, 9, size=(2, 5))
Out[]:
array([[5, 7, 4, 6, 4],
       [6, 8, 8, 5, 3]])
```

- `random(size=None)` returns a random float value between 0 and 1 which is drawn from the *continuous uniform* distribution.

```
# Generating a random float
In []: np.random.random()
Out[]: 0.6013749764953444

# Generating a one-dimensional array
```

```
In []: np.random.random(3)
Out[]: array([0.69929315, 0.61152299, 0.91313813])

# Generating a two-dimensional array
In []: np.random.random((3, 2))
Out[]:
array([[0.55779547, 0.6822698 ],
       [0.75476145, 0.224952  ],
       [0.99264158, 0.02755453]])
```

- `binomial(n, p, size=None)` returns samples drawn from a binomial distribution with n trials and p probability of success where n is greater than 0 and p is in the interval of 0 and 1.

```
# Number of trials, probability of each trial
In []: n, p = 1, .5

# Flipping a coin 1 time for 50 times
In []: samples = np.random.binomial(n, p, 50)

In []: samples
Out[]:
array([1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
       0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1,
       0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0])
```

- `normal(mu=0.0, sigma=1.0, size=None)` draws random samples from a normal (Gaussian) distribution. If no arguments provided, a sample will be drawn from N(0, 1).

```
# Initialize mu and sigma
In []: mu, sigma = 0, 0.1

# Drawing 5 samples in a one-dimensional array
In []: np.random.normal(mu, sigma, 5)
Out[]: array([ 0.06790522, 0.0092956, 0.07063545,
               0.28022021, -0.13597963])

# Drawing 10 samples in a two-dimensional array of
```

```
# shape (2, 5)
In []: np.random.normal(mu, sigma, (2, 5))
Out[]:
array([[-0.10696306, -0.0147926, -0.07027478, 0.04399432,
        -0.03861839],
       [-0.02004485, 0.08760261, 0.18348247, -0.09351321,
        -0.19487115]])
```

- `uniform(low=0.0, high=1.0, size=None)` draws samples from a uniform distribution over the interval 0 (including) and 1 (excluding), if no arguments are provided. In other words, any value drawn is equally likely within the interval.

```
# Creating a one-dimensional array with samples drawn
# within [-1, 0)
In []: np.random.uniform(-1, 0, 10)
Out[]:
array([-0.7910379, -0.64144624, -0.64691011, -0.03817127,
       -0.24480339, -0.82549031, -0.37500955, -0.88304322,
       -0.35196588, -0.51377252])

# Creating a two-dimensional array with samples drawn
# within [0, 1)
In []: np.random.uniform(size=(5, 2))
Out[]:
array([[0.43155784, 0.41360889],
       [0.81813931, 0.70115211],
       [0.40003811, 0.2114227 ],
       [0.95487774, 0.92251769],
       [0.91042434, 0.98697917]])
```

In addition to functions shown above, we can draw samples from various other distributions such as Poisson, Gamma, Exponential, etc. using NumPy.

10.4 Array Attributes and Methods

We now have some idea about the working of NumPy arrays. Let us now explore the functionalities provided by them. As with any Python object,

NumPy arrays also have a rich set of attributes and methods which simplifies the data analysis process to a great extent. Following are the most useful array attributes. For illustration purpose, we will be using previously defined arrays.

- `ndim` attribute displays the number of dimensions of an array. Using this attribute on `n_call_vol` and `pcr`, we expect dimensions to be 1 and 2 respectively. Let's check.

```
# Checking dimensions for n_call_vol array
In []: n_call_vol.ndim
Out[]: 1

In []: n_2d.ndim
Out[]: 2
```

- `shape` returns a tuple with the dimensions of the array. It may also be used to reshape the array in-place by assigning a tuple of array dimensions to it.

```
# Checking the shape of the one-dimensional array
In []: n_put_vol.shape
Out[]: (5,)

# Checking shape of the two-dimensional array
In []: n_2d.shape
Out[]: (2, 5) # It shows 2 rows and 5 columns

# Printing n_2d with 2 rows and 5 columns
In []: n_2d
Out[]:
array([[52.89, 45.14, 63.84, 77.1 , 74.6 ],
       [49.51, 50.45, 59.11, 80.49, 65.11]])

# Reshaping n_2d using the shape attribute
In []: n_2d.shape = (5, 2)

# Printing reshaped array
In []: n_2d
```

```
Out[]:
array([[52.89, 45.14],
       [63.84, 77.1 ],
       [74.6 , 49.51],
       [50.45, 59.11],
       [80.49, 65.11]])
```

- `size` returns the number of elements in the array.

```
In []: n_call_vol.size
Out[]: 5

In []: n_2d.size
Out[]: 10
```

- `dtype` returns the data-type of the array's elements. As we learned above, NumPy comes with its own data type just like regular built-in data types such as int, float, str, etc.

```
In []: n_put_vol.dtype
Out[]: dtype('float64')
```

A typical first step in analyzing a data is getting to the data in the first place. In an ideal data analysis process, we generally have thousands of numbers which need to be analyzed. Simply staring at these numbers won't provide us with any insights. Instead, what we can do is generate summary statistics of the data. Among many useful features, NumPy also provides various statistical functions which are good to perform such statistics on arrays.

Let us create a `samples` array and populate it with samples drawn from a normal distribution with a mean of 5 and standard deviation of 1.5 and compute various statistics on it.

```
# Creating a one-dimensional array with 1000 samples drawn
# from a normal distribution
In []: samples = np.random.normal(5, 1.5, 1000)

# Creating a two-dimensional array with 25 samples
```

```
# drawn from a normal distribution
In []: samples_2d = np.random.normal(5, 1.5, size=(5, 5))

In []: samples_2d
Out[]:
array([[5.30338102, 6.29371936, 2.74075451, 3.45505812,
        7.24391809],
       [5.20554917, 5.33264245, 6.08886915, 5.06753721,
        6.36235494],
       [5.86023616, 5.54254211, 5.38921487, 6.77609903,
        7.79595902],
       [5.81532883, 0.76402556, 5.01475416, 5.20297957,
        7.57517601],
       [5.76591337, 1.79107751, 5.03874984, 5.05631362,
        2.16099478]])
```

- `mean(a, axis=None)` returns the average of the array elements. The average is computed over the flattened array by default, otherwise over the specified axis.

- `average(a, axis=None)` returns the average of the array elements and works similar to that of `mean()`.

```
# Computing mean
In []: np.mean(samples)
Out[]: 5.009649198007546

In []: np.average(samples)
Out[]: 5.009649198007546

# Computing mean with axis=1 (over each row)
In []: np.mean(samples_2d, axis=1)
Out[]: array([5.00736622, 5.61139058, 6.27281024,
              4.87445283, 3.96260983])

In []: np.average(samples_2d, axis=1)
Out[]: array([5.00736622, 5.61139058, 6.27281024,
              4.87445283, 3.96260983])
```

- `max(a, axis=None)` returns the maximum of an array or maximum along an axis.

```
In []: np.max(samples)
Out[]: 9.626572532562523

In []: np.max(samples_2d, axis=1)
Out[]: array([7.24391809, 6.36235494, 7.79595902,
              7.57517601, 5.76591337])
```

- `median(a, axis=None)` returns the median along the specified axis.

```
In []: np.median(samples)
Out[]: 5.0074934668143865

In []: np.median(samples_2d)
Out[]: 5.332642448141249
```

- `min(a, axis=None)` returns the minimum of an array or minimum along an axis.

```
In []: np.min(samples)
Out[]: 0.1551821703754115

In []: np.min(samples_2d, axis=1)
Out[]: array([2.74075451, 5.06753721, 5.38921487,
              0.76402556, 1.79107751])
```

- `var(a, axis=None)` returns the variance of an array or along the specified axis.

```
In []: np.var(samples)
Out[]: 2.2967299389550466

In []: np.var(samples_2d)
Out[]: 2.93390175942658

# The variance is computed over each column of numbers
In []: np.var(samples_2d, axis=0)
Out[]: array([0.07693981, 4.95043105, 1.26742732,
              1.10560727, 4.37281009])
```

- `std(a, axis=None)` returns the standard deviation of an array or along the specified axis.

```
In [ ]: np.std(samples)
Out[ ]: 1.5154965981337756

In [ ]: np.std(samples_2d)
Out[ ]: 1.7128636137844075
```

- `sum(a, axis=None)` returns the sum of array elements.

```
# Recalling the array n_put_vol
In [ ]: n_put_vol
Out[ ]: array([52.89, 45.14, 63.84, 77.1 , 74.6 ])

# Computing sum of all elements within n_put_vol
In [ ]: np.sum(n_put_vol)
Out[ ]: 313.57

# Computing sum of all array over each row
In [ ]: np.sum(samples_2d, axis=1)
Out[ ]: array([25.03683109, 28.05695291, 31.36405118,
               24.37226413, 19.81304913])
```

- `cumsum(a, axis=None)` returns the cumulative sum of the elements along a given axis.

```
In [ ]: np.cumsum(n_put_vol)
Out[ ]: array([ 52.89,  98.03, 161.87, 238.97, 313.57])
```

The methods discussed above can also be directly called upon NumPy objects such as `samples`, `n_put_vol`, `samples_2d`, etc. instead of using the `np.` format as shown below. The output will be the same in both cases.

```
# Using np. format to compute the sum
In [ ]: np.sum(samples)
Out[ ]: 5009.649198007546

# Calling sum() directly on a NumPy object
In [ ]: samples.sum()
Out[ ]: 5009.649198007546
```

10.5 Array Manipulation

NumPy defines a new data type called `ndarray` for the array object it creates. This also means that various operators such as arithmetic operators, logical operator, boolean operators, etc. work in ways unique to it as we've seen so far. There's a flexible and useful array manipulation technique that NumPy provides to use on its data structure using *broadcasting*.

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations (with certain constraints). The smaller array is *'broadcast'* across the larger array so that they have compatible shapes. It also provides a mean of vectorizing array operations.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape as in the following example.

```
In []: a = np.array([1, 2, 3])

In []: b = np.array([3, 3, 3])

In []: a * b
Out[]: array([3, 6, 9])
```

NumPy's broadcasting rule relaxes this constraint when the array's shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in operation as depicted below:

```
In []: a = np.array([1, 2, 3])

In []: b = 3

In []: a * b
Out[]: array([3, 6, 9])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` in the above example being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. Here, the

stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

The code in the last example is more efficient because broadcasting moves less memory around during the multiplication than that of its counterpart defined above it. Along with efficient number processing capabilities, NumPy also provides various methods for array manipulation thereby proving versatility. We discuss some of them here.

- `exp(*args)` returns the exponential of all elements in the input array. The numbers will be raised to e also known as Euler's number.

```
# Computing exponentials for the array 'a'
In []: np.exp(a)
Out[]: array([ 2.71828183, 7.3890561, 20.08553692])
```

- `sqrt(*args)` returns the positive square-root of an array, element-wise.

```
# Computing square roots of a given array
In []: np.sqrt([1, 4, 9, 16, 25])
Out[]: array([1., 2., 3., 4., 5.])
```

- `reshape(new_shape)` gives a new shape to an array without changing its data.

```
# Creating a one-dimensional array with 12 elements
In []: res = np.arange(12)

In []: res
Out[]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

# Reshaping the 'res' array to 2-dimensional array
In []: np.reshape(res, (3, 4))
Out[]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
# Reshaping the dimensions from (3, 4) to (2, 6)
In []: np.reshape(res, (2, 6))
Out[]:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

- `resize(a, new_shape)` return a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copies of *a*.

```
# Creating a one-dimensional array
In []: demo = np.arange(4)

In []: demo
Out[]: array([0, 1, 2, 3])

# Resizing a 'demo' array to (2, 2)
In []: np.resize(demo, (2, 2))
Out[]:
array([[0, 1],
       [2, 3]])

# Resizing a 'demo' greater than its size.
In []: np.resize(demo, (4, 2))
Out[]:
array([[0, 1],
       [2, 3],
       [0, 1],
       [2, 3]])
```

- `round(a, decimals=0)` round an array to the given number of decimals. If decimals are not given, elements will be rounded to the whole number.

```
# Creating a one-dimensional array
In []: a = np.random.rand(5)

# Printing array
```

```
In []: a
Out[]: array([0.71056952, 0.58306487, 0.13270092,
              0.38583513, 0.7912277 ])

# Rounding to 0 decimals
In []: a.round()
Out[]: array([1., 1., 0., 0., 1.])

# Rounding to 0 decimals using the np.round syntax
In []: np.round(a)
Out[]: array([1., 1., 0., 0., 1.])

# Rounding to 2 decimals
In []: a.round(2)
Out[]: array([0.71, 0.58, 0.13, 0.39, 0.79])

# Rounding to 3 decimals using the np.round syntax
In []: np.round(a, 3)
Out[]: array([0.711, 0.583, 0.133, 0.386, 0.791])
```

- `sort(a, kind='quicksort')` returns a sorted copy of an array. The default sorting algorithm used is *quicksort*. Other available options are *mergesort* and *heapsort*.

```
In []: np.sort(n_put_vol)
Out[]: array([45.14, 52.89, 63.84, 74.6 , 77.1 ])

In []: np.sort(samples_2d)
Out[]:
array([[2.74075451, 3.45505812, 5.30338102, 6.29371936,
        7.24391809],
       [5.06753721, 5.20554917, 5.33264245, 6.08886915,
        6.36235494],
       [5.38921487, 5.54254211, 5.86023616, 6.77609903,
        7.79595902],
       [0.76402556, 5.01475416, 5.20297957, 5.81532883,
        7.57517601],
       [1.79107751, 2.16099478, 5.03874984, 5.05631362,
        5.76591337]])
```

- `vstack(tup)` stacks arrays provided via *tup* in sequence vertically (row wise).
- `hstack(tup)` stacks arrays provided via *tup* in sequence horizontally (column wise).
- `column_stack(tup)` stacks 1-dimensional arrays as column into a 2-dimensional array. It takes a sequence of 1-D arrays and stacks them as columns to make a single 2-D array.

```
# Creating sample arrays
In []: a = np.array([1, 2, 3])

In []: b = np.array([4, 5, 6])

In []: c = np.array([7, 8, 9])

# Stacking arrays vertically
In []: np.vstack((a, b, c))
Out[]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

# Stacking arrays horizontally
In []: np.hstack((a, b, c))
Out[]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Stacking two arrays together
In []: np.column_stack((a, b))
Out[]:
array([[1, 4],
       [2, 5],
       [3, 6]])

# Stacking three arrays together
In []: np.column_stack((a, b, c))
Out[]:
array([[1, 4, 7],
```

```
       [2, 5, 8],
       [3, 6, 9]])
```

- `transpose()` permutes the dimensions of an array.

```
# Creating a two-dimensional array with shape (2, 4)
In []: a = np.arange(8).reshape(2, 4)

# Printing it
In []: a
Out[]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

# Transposing the array
In []: a.transpose()
Out[]:
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

10.6 Array Indexing and Iterating

NumPy is an excellent library for efficient number crunching along with ease of use. It seamlessly integrates with Python and its syntax. Following this attribute, NumPy provides subsetting and iterating techniques very similar to lists. We can use square brackets to subset NumPy arrays, Python built-in constructs to iterate, and other built-in methods to slice them.

10.6.1 Indexing and Subsetting

NumPy arrays follow indexing structure similar to Python lists. Index starts with 0 and each element in an array is associated with a unique index. Below table shows NumPy indexing for a one-dimensional array.

Index	**0**	**1**	**2**	**3**	**4**
np.array	52	88	41	63	94

The index structure for a two-dimensional array with a shape of (3, 3) is shown below.

Index	0	1	2
0	a	b	c
1	d	e	f
2	g	h	i

The two arrays `arr_1d` and `arr_2d` which depicts the above-shown structure have been created below:

```
# Creating one-dimensional array
In []: arr_1d = np.array([52, 88, 41, 63, 94])

# Creating two-dimensional array
In []: arr_2d = np.array([['a', 'b', 'c'],
  ...:                    ['d', 'e', 'f'],
  ...:                    ['g', 'h', 'i']])
```

We use square brackets `[]` to subset each element from NumPy arrays. Let us subset arrays created above using indexing.

```
# Slicing the element at index 0
In []: arr_1d[0]
Out[]: 52

# Slicing the last element using negative index
In []: arr_1d[-1]
Out[]: 94

# Slicing elements from position 2 (inclusive) to
# 5 (exclusive)
In []: arr_1d[2:5]
Out[]: array([41, 63, 94])
```

In the above examples, we sliced a one-dimensional array. Similarly, square brackets also allow slicing two-dimensional using the syntax `[r, c]` where *r* is a row and *c* is a column.

```
# Slicing the element at position (0, 1)
In []: arr_2d[0, 1]
Out[]: 'b'

# Slicing the element at position (1, 2)
In []: arr_2d[1, 2]
Out[]: 'f'

# Slicing the element at position (2, 0)
In []: arr_2d[2, 0]
Out[]: 'g'

# Slicing the first row
In []: arr_2d[0, ]
Out[]: array(['a', 'b', 'c'], dtype='<U1')

# Slicing the last column
In []: arr_2d[:, 2]
Out[]: array(['c', 'f', 'i'], dtype='<U1')
```

Notice the syntax in the last example where we slice the last column. The : has been provided as an input which denotes all elements and then filtering the last column. Using only : would return us all elements in the array.

```
In []: arr_2d[:]
Out[]:
array([['a', 'b', 'c'],
       ['d', 'e', 'f'],
       ['g', 'h', 'i']], dtype='<U1')
```

10.6.2 Boolean Indexing

NumPy arrays can be indexed with other arrays (or lists). The arrays used for indexing other arrays are known as *index arrays*. Mostly, it is a simple array which is used to subset other arrays. The use of index arrays ranges from simple, straightforward cases to complex and hard to understand cases. When an array is indexed using another array, a copy of the original data is returned, not a view as one gets for slices. To illustrate:

```
# Creating an array
In []: arr = np.arange(1, 10)

# Printing the array
In []: arr
Out[]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Subsetting the array `arr` using an anonymous array
In []: arr[np.array([2, 5, 5, 1])]
Out[]: array([3, 6, 6, 2])
```

We create an array `arr` with ten elements in the above example. Then we try to subset it using an anonymous index array. The index array consisting of the values 2, 5, 5 and 1 correspondingly create an array of length 4, i.e. same as the length index array. Values in the index array work as an index to subset (in the above-given operation) and it simply returns the corresponding values from the `arr`.

Extending this concept, an array can be indexed with itself. Using logical operators, NumPy arrays can be filtered as desired. Consider a scenario, where we need to filter array values which are greater than a certain threshold. This is shown below:

```
# Creating an random array with 20 values
In []: rand_arr = np.random.randint(1, 50, 20)

# Printing the array
In []: rand_arr
Out[]:
array([14, 25, 39, 18, 40, 10, 33, 36, 29, 25, 27,  4, 28,
       43, 43, 19, 30,29, 47, 41])

# Filtering the array values which are greater than 30
In []: rand_arr[rand_arr > 30]
Out[]: array([39, 40, 33, 36, 43, 43, 47, 41])
```

Here, we create an array with the name `rand_arr` with 20 random values. We then try to subset it with values which are greater than 30 using the logical operator >. When an array is being sliced using the logical operator,

NumPy generates an anonymous array of `True` and `False` values which is then used to subset the array. To illustrate this, let us execute the code used to subset the `rand_arr`, i.e. code written within the square brackets.

```
In []: filter_ = rand_arr > 30

In []: filter_
Out[]:
array([False, False,  True, False,  True, False,  True,
       True, False, False, False, False, False,  True,
       True, False, False, False, True,  True])
```

It returned a boolean array with only `True` and `False` values. Here, `True` appears wherever the logical condition holds true. NumPy uses this outputted array to subset the original array and returns only those values where it is `True`.

Apart from this approach, NumPy provides a `where` method using which we can achieve the same filtered output. We pass a logical condition within `where` condition, and it will return an array with all values for which conditions stands true. We filter out all values greater than 30 from the `rand_arr` using the `where` condition in the following example:

```
# Filtering an array using np.where method
In []: rand_arr[np.where(rand_arr > 30)]
Out[]: array([39, 40, 33, 36, 43, 43, 47, 41])
```

We got the same result by executing `rand_arr[rand_arr > 30]` and `rand_arr[np.where(rand_arr > 30)]`. However, the `where` method provided by NumPy just do not filter values. Instead, it can be used for more versatile scenarios. Below given is the official syntax:

```
np.where[condition[, x, y]]
```

It returns the elements, either from `x` or `y`, depending on `condition`. As these parameters, `x` and `y` are optional, `condition` when true yields `x` if given or boolean `True`, otherwise `y` or boolean `False`.

Below we create an array `heights` that contains 20 elements with height ranging from 150 to 160 and look at various uses of `where` method.

```
# Creating an array heights
In []: heights = np.random.uniform(150, 160, size=20)

In []: heights
Out[]:
array([153.69911134, 154.12173942, 150.35772942,
       151.53160722, 153.27900307, 154.42448961,
       153.25276742, 151.08520803, 154.13922276,
       159.71336708, 151.45302507, 155.01280829,
       156.9504274 , 154.40626961, 155.46637317,
       156.36825413,151.5096344 , 156.75707004,
       151.14597394, 153.03848597])
```

Usage 1: Without `x` and `y` parameters. Using the `where` method without the optional parameter as illustrated in the following example would return the index values of the original array where the condition is true.

```
In []: np.where(heights > 153)
Out[]:
(array([ 0, 1, 4, 5, 6, 8, 9, 11, 12, 13, 14, 15, 17, 19],
       dtype=int64),)
```

The above codes returned index values of the `heights` array where values are greater than 153. This scenario is very similar to the one we have seen above with the random array `rand_arr` where we tried to filter values above 30. Here, the output is merely the index values. If we want the original values, we need to subset the `heights` array using the output that we obtained.

Usage 2: With `x` as `True` and `y` as `False`. Having these optional parameters in place would return either of the parameters based on the condition. This is shown in the below example:

```
In []: np.where(heights > 153, True, False)
Out[]:
array([True, True, False, False, True, True, True, False,
       True, True, False, True, True, True, True, True,
       False, True,False, True])
```

The output in the *Usage 2* provides either `True` or `False` for all the elements in the `heights` array in contrast to the *Usage 1* where it returned index values of only those elements where the condition was true. The optional parameters can also be array like elements instead of scalars or static value such as `True` or `False`.

Usage 3: With `x` and `y` being arrays. Now that we have quite a good understanding of how the `where` method works, it is fairly easy to guess the output. The output will contain values from either `x` array or `y` array based on the condition in the first argument. For example:

```
# Creating an array 'x_array'
In []: x_array = np.arange(11, 31, 1)

In []: x_array
Out[]:
array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
       24, 25, 26, 27, 28, 29, 30])

# Creating an array 'y_array'
In []: y_array = np.arange(111, 131, 1)

In []: y_array
Out[]:
array([111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
       121, 122, 123, 124, 125, 126, 127, 128, 129, 130])

In []: np.where(heights > 153, x_array, y_array)
Out[]:
array([ 11,  12, 113, 114,  15,  16,  17, 118,  19,  20,
       121,  22,  23, 24,  25,  26, 127,  28, 129,  30])
```

As expected, the output of the above code snippet contains values from the array `x_array` when the value in the `heights` array is greater than 153, otherwise, the value from the `y_array` will be outputted.

Having understood the working of `where` method provided by the NumPy library, let us now see how it is useful in back-testing strategies. Consider a scenario where we have all the required data for generating trading signals.

Data that we have for this hypothetical example is the close price of a stock for 20 periods and its average price.

```
# Hypothetical close prices for 20 periods
In []: close_price = np.random.randint(132, 140, 20)

# Printing close_price
In []: close_price
Out[]:
array([137, 138, 133, 132, 134, 139, 132, 138, 137, 135,
       136, 134, 134, 139, 135, 133, 136, 139, 132, 134])
```

We are to generate trading signals based on the buy condition given to us. i.e. we go long or buy the stock when the closing price is greater than the average price of 135.45. It can be easily computed using the `where` method as shown below:

```
# Average close price
In []: avg_price = 135.45

# Computing trading signals with 1 being 'buy' and 0
# represents 'no signal'
In []: signals = np.where(close_price > avg_price, 1, 0)

# Printing signals
In []: signals
Out[]: array([1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0,
              0, 1, 1, 0, 0])
```

The `signals` array contains the trading signals where `1` represents the `buy` and `0` represents no trading signal.

10.6.3 Iterating Over Arrays

NumPy arrays are iterable objects in Python which means that we can directly iterate over them using the `iter()` and `next()` methods as with any other iterable. This also implies that we can use built-in looping constructs to iterate over them. The following examples show iterating NumPy arrays using a `for` loop.

```
# Looping over a one-dimensional array
In []: for element in arr_1d:
  ...:     print(element)

# Output
52
88
41
63
94
```

Looping over a one-dimensional array is easy and straight forward. But, if we are to execute the `for` loop with `arr_2d`, it will be traverse all rows and provide that as the output. It is demonstrated in the following example.

```
In []: for element in arr_2d:
  ...:     print(element)

# Output
['a' 'b' 'c']
['d' 'e' 'f']
['g' 'h' 'i']
```

To iterate over two-dimensional arrays, we need to take care of both axes. A separate `for` loop can be used in a nested format to traverse through each element as shown below:

```
In []: for element in arr_2d:
  ...:     for e in element:
  ...:         print(e)

# Output
a
b
c
d
e
f
g
```

```
h
i
```

The output that we got can also be achieved using `nditer()` method of NumPy, and it works for irrespective of dimensions.

```
In []: for element in np.nditer(arr_2d):
  ...:     print(element)

# Output
a
b
c
d
e
f
g
h
i
```

This brings us to the end of a journey with the NumPy module. The examples provided above depicts only a minimal set of NumPy functionalities. Though not comprehensive, it should give us a pretty good feel about what is NumPy and why we should be using it.

10.7 Key Takeaways

1. NumPy library is used to perform scientific computing in Python.
2. It is not a part of the Python Standard Library and needs to be installed explicitly before it can be used in a program.
3. It allows creating n-dimensional arrays of the type `ndarray`.
4. NumPy arrays can hold elements of single data type only.
5. They can be created using any sequential data structures such as lists or tuples, or using built-in NumPy functions.
6. The random module of the NumPy library allows generating samples from various data distributions.
7. NumPy supports for element-wise operation using broadcast functionality.

8. Similar to lists, NumPy arrays can also be sliced using square brackets `[]` and starts indexing with `0`.
9. It is also possible to slice NumPy arrays based on logical conditions. The resultant array would be an array of boolean True or False based on which other arrays are sliced or filtered. This is known as boolean indexing.

Chapter 11

Pandas Module

Pandas is a Python library to deal with sequential and tabular data. It includes many tools to manage, analyze and manipulate data in a convenient and efficient manner. We can think of its data structures as akin to database tables or spreadsheets.

Pandas is built on top of the Numpy library and has two primary data structures viz. Series (1-dimensional) and DataFrame (2- dimensional). It can handle both homogeneous and heterogeneous data, and some of its many capabilities are:

- ETL tools (Extraction, Transformation and Load tools)
- Dealing with missing data (NaN)
- Dealing with data files (csv, xls, db, hdf5, etc.)
- Time-series manipulation tools

In the Python ecosystem, Pandas is the best choice to retrieve, manipulate, analyze and transform financial data.

11.1 Pandas Installation

The official documentation[1] has a detailed explanation that spans over several pages on installing Pandas. We summarize it below.

[1] https://pandas.pydata.org/pandas-docs/stable/install.html

11.1.1 Installing with `pip`

The simplest way to install Pandas is from PyPI.
In a terminal window, run the following command.

```
pip install pandas
```

In your code, you can use the escape character '**!**' to install pandas directly from your Python console.

```
!pip install pandas
```

Pip is a useful tool to manage Python's packages and it is worth investing some time in knowing it better.

```
pip help
```

11.1.2 Installing with Conda environments

For advanced users, who like to work with Python environments for each project, you can create a new environment and install pandas as shown below.

```
conda create -n EPAT python
source activate EPAT
conda install pandas
```

11.1.3 Testing Pandas installation

To check the installation, Pandas comes with a test suite to test almost all of the codebase and verify that everything is working.

```
import pandas as pd
pd.test()
```

11.2 What problem does Pandas solve?

Pandas works with homogeneous data series (1-Dimension) and heterogeneous tabular data series (2-Dimensions). It includes a multitude of tools to work with these data types, such as:

- Indexes and labels.
- Searching of elements.
- Insertion, deletion and modification of elements.
- Apply set techniques, such as grouping, joining, selecting, etc.
- Data processing and cleaning.
- Work with time series.
- Make statistical calculations
- Draw graphics
- Connectors for multiple data file formats, such as, csv, xlsx, hdf5, etc.

11.3 Pandas Series

The first data structure in Pandas that we are going to see is the Series. They are homogeneous one-dimensional objects, that is, all data are of the same type and are implicitly labeled with an index.

For example, we can have a Series of integers, real numbers, characters, strings, dictionaries, etc. We can conveniently manipulate these series performing operations like adding, deleting, ordering, joining, filtering, vectorized operations, statistical analysis, plotting, etc.

Let's see some examples of how to create and manipulate a Pandas Series:

- We will start by creating an empty Pandas Series:

```
import pandas as pd
s = pd.Series()
print(s)

Out[]: Series([], dtype: float64)
```

- Let's create a Pandas Series of integers and print it:

```
import pandas as pd
s = pd.Series([1, 2, 3, 4, 5, 6, 7])
print(s)

Out[]:  0    1
```

```
1    2
2    3
3    4
4    5
5    6
6    7
dtype: int64
```

- Let's create a Pandas Series of characters:

```
import pandas as pd
s = pd.Series(['a', 'b', 'c', 'd', 'e'])
print(s)

Out[]:  0    a
        1    b
        2    c
        3    d
        4    e
        dtype: object
```

- Let's create a random Pandas Series of float numbers:

```
import pandas as pd
import numpy as np
s = pd.Series(np.random.randn(5))
print(s)

Out[]:  0    0.383567
        1    0.869761
        2    1.100957
        3   -0.259689
        4    0.704537
        dtype: float64
```

In all these examples, we have allowed the index label to appear by default (without explicitly programming it). It starts at 0, and we can check the index as:

```
In []: s.index

Out[]: RangeIndex(start=0, stop=5, step=1)
```

But we can also specify the index we need, for example:

```
In []: s = pd.Series(np.random.randn(5),
                     index=['a', 'b', 'c', 'd', 'e'])

Out[]: a    1.392051
       b    0.515690
       c   -0.432243
       d   -0.803225
       e    0.832119
       dtype: float64
```

- Let's create a Pandas Series from a dictionary:

```
import pandas as pd
dictionary = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}
s = pd.Series(dictionary)
print(s)

Out[]: a    1
       b    2
       c    3
       d    4
       e    5
       dtype: int64
```

In this case, the Pandas Series is created with the dictonary keys as index unless we specify any other index.

11.3.1 Simple operations with Pandas Series

When we have a Pandas Series, we can perform several simple operations on it. For example, let's create two Series. One from a dictionary and the other from an array of integers:

```
In []: import pandas as pd
       dictionary = {'a' : 1, 'b' : 2, 'c' : 3, 'd': 4,
                     'e': 5}
       s1 = pd.Series(dictionary)

       array = [1, 2, 3, 4, 5]
       s2 = pd.Series(array)

Out[]: a    1
       b    2
       c    3
       d    4
       e    5
       dtype: int64

       0    1
       1    2
       2    3
       3    4
       4    5
       dtype: int64
```

We can perform operations similar to Numpy arrays:

- Selecting one item from the Pandas Series by means of its index:

```
In []: s1[0] # Select the first element
Out[]: 1

In []: s1['a']
Out[]: 1

In []: s2[0]
Out[]: 1
```

- Selecting several items from the Pandas Series by means of its index:

```
In []: s1[[1, 4]]
Out[]: b    2
```

```
        e    5
        dtype: int64

In []: s1[['b', 'e']]
Out[]: b    2
        e    5
         dtype: int64

In []: s2[[1, 4]]
Out[]: b    2
        e    5
         dtype: int64
```

- Get the series starting from an element:

```
In []: s1[2:]
Out[]: c    3
        d    4
        e    5
        dtype: int64

In []: s2[2:]
Out[]: 2    3
        3    4
        4    5
        dtype: int64
```

- Get the series up to one element:

```
In []: s1[:2]
Out[]: a    1
        b    2
        dtype: int64

In []: s2[:2]
Out[]: 0    1
        1    2
        dtype: int64
```

We can perform operations like a dictionary:

- Assign a value:

```
In []: s1[1] = 99
       s1['a'] = 99

Out[]: a     1
       b    99
       c     3
       d     4
       e     5
       dtype: int64

In []: s2[1] = 99
       print(s2)

Out[]: 0     1
       1    99
       2     3
       3     4
       4     5
       dtype: int64
```

- Get a value by index (like dictionary key):

```
In []: s.get('b')
Out[]: 2
```

Here are some powerful vectorized operations that let us perform quickly calculations, for example:

- Add, subtract, multiply, divide, power, and almost any NumPy function that accepts NumPy arrays.

```
s1 + 2
s1 - 2
s1 * 2
s1 / 2
s1 ** 2
np.exp(s1)
```

- We can perform the same operations over two Pandas Series although these must be aligned, that is, to have the same index, in other case, perform a Union operation.

```
In []: s1 + s1 # The indices are aligned
Out[]: a     2
       b     4
       c     6
       d     8
       e    10
       dtype: int64

In []: s1 + s2 # The indices are unaligned
Out[]: a   NaN
       b   NaN
       c   NaN
       d   NaN
       e   NaN
       0   NaN
       1   NaN
       2   NaN
       3   NaN
       4   NaN
       dtype: float64
```

11.4 Pandas DataFrame

The second data structure in Pandas that we are going to see is the DataFrame.

Pandas DataFrame is a heterogeneous two-dimensional object, that is, the data are of the same type within each column but it could be a different data type for each column and are implicitly or explicitly labeled with an index.

We can think of a DataFrame as a database table, in which we store heterogeneous data. For example, a DataFrame with one column for the first name, another for the last name and a third column for the phone

number, or a dataframe with columns to store the opening price, close price, high, low, volume, and so on.

The index can be implicit, starting with zero or we can specify it ourselves, even working with dates and times as indexes as well. Let's see some examples of how to create and manipulate a Pandas DataFrame.

- Creating an empty DataFrame:

```
In []: import pandas as pd
       s = pd.DataFrame()
       print(s)

Out[]: Empty DataFrame
       Columns: []
       Index: []
```

- Creating an empty structure DataFrame:

```
In []: import pandas as pd
       s = pd.DataFrame(columns=['A', 'B', 'C', 'D', 'E'])
       print(s)

Out[]: Empty DataFrame
       Columns: [A, B, C, D, E]
       Index: []

In []: import pandas as pd
       s = pd.DataFrame(columns=['A', 'B', 'C', 'D', 'E'],
                        index=range(1, 6))
       print(s)

Out[]:      A    B    C    D    E
       1  NaN  NaN  NaN  NaN  NaN
       2  NaN  NaN  NaN  NaN  NaN
       3  NaN  NaN  NaN  NaN  NaN
       4  NaN  NaN  NaN  NaN  NaN
       5  NaN  NaN  NaN  NaN  NaN
```

- Creating a DataFrame passing a NumPy array:

```
In []: array =  {'A' : [1, 2, 3, 4],
                 'B' : [4, 3, 2, 1]}

       pd.DataFrame(array)

Out[]:   A   B
       0 1   4
       1 2   3
       2 3   2
       3 4   1
```

- Creating a DataFrame passing a NumPy array, with datetime index:

```
In []: import pandas as pd
       array = {'A':[1, 2, 3, 4], 'B':[4, 3, 2, 1]}
       index = pd.DatetimeIndex(['2018-12-01',
                                 '2018-12-02',
                                 '2018-12-03',
                                 '2018-12-04'])
       pd.DataFrame(array, index=index)

Out[]:              A   B
       2018-12-01   1   4
       2018-12-02   2   3
       2018-12-03   3   2
       2018-12-04   4   1
```

- Creating a DataFrame passing a Dictionary:

```
In []: import pandas as pd
       dictionary = {'a':1, 'b':2, 'c':3, 'd':4, 'e': 5}
       pd.DataFrame([dictionary])

Out[]:   a   b   c   d   e
       0 1   2   3   4   5
```

- Viewing a DataFrame: We can use some methods to explore the Pandas DataFrame:

First, we go to create a Pandas DataFrame to work with it.

```
In []: import pandas as pd
       pd.DataFrame({'A':np.random.randn(10),
                     'B':np.random.randn(10),
                     'C':np.random.randn(10)})
```

```
Out[]:      A           B           C
       0   0.164358    1.689183    1.745963
       1  -1.830385    0.035618    0.047832
       2   1.304339    2.236809    0.920484
       3   0.365616    1.877610   -0.287531
       4  -0.741372   -1.443922   -1.566839
       5  -0.119836   -1.249112   -0.134560
       6  -0.848425   -0.569149   -1.222911
       7  -1.172688    0.515443    1.492492
       8   0.765836    0.307303    0.788815
       9   0.761520   -0.409206    1.298350
```

- Get the first three rows:

```
In []: import pandas as pd
       df = pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
       df.head(3)
```

```
Out[]:      A           B           C
       0   0.164358    1.689183    1.745963
       1  -1.830385    0.035618    0.047832
       2   1.304339    2.236809    0.920484
```

- Get the last three rows:

```
In []: import pandas as pd
       df = pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
       df.tail(3)
```

```
Out[]:      A          B          C
        7  -1.172688   0.515443   1.492492
        8   0.765836   0.307303   0.788815
        9   0.761520  -0.409206   1.298350
```

- Get the DataFrame's index:

```
In []: import pandas as pd
       df = pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
       df.index

Out[]: RangeIndex(start=0, stop=10, step=1)
```

- Get the DataFrame's columns:

```
In []: import pandas as pd
       df = pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
       df.columns

Out[]: Index(['A', 'B', 'C'], dtype='object')
```

- Get the DataFrame's values:

```
In []: import pandas as pd
       df = pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
       df.values

Out[]: array([[ 0.6612966 , -0.60985049,  1.11955054],
              [-0.74105636,  1.42532491, -0.74883362],
              [ 0.10406892,  0.5511436 ,  2.63730671],
              [-0.73027121, -0.11088373, -0.19143175],
              [ 0.11676573,  0.27582786, -0.38271609],
              [ 0.51073858, -0.3313141 ,  0.20516165],
```

```
[ 0.23917755,  0.55362   , -0.62717194],
[ 0.25565784, -1.4960713 ,  0.58886377],
[ 1.20284041,  0.21173483,  2.0331718 ],
[ 0.62247283,  2.18407105,  0.02431867]])
```

11.5 Importing data in Pandas

Pandas DataFrame is able to read several data formats, some of the most used are: CSV, JSON, Excel, HDF5, SQL, etc.

11.5.1 Importing data from CSV file

One of the most useful functions is `read_csv` that allows us to read csv files with almost any format and load it into our DataFrame to work with it. Let's see how to work with csv files:

```
import pandas as pd
df = pd.read_csv('Filename.csv')
type(df)

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

This simple operation, loads the csv file into the Pandas DataFrame after which we can explore it as we have seen before.

11.5.2 Customizing pandas import

Sometimes the format of the csv file come with a particular separator or we need specific columns or rows. We will now see some ways to deal with this.

In this example, we want to load a csv file with blank space as separator:

```
import pandas as pd
df = pd.read_csv('Filename.csv', sep=' ')
```

In this example, we want to load columns from 0 and 5 and the first 100 rows:

```
import pandas as pd
df = pd.read_csv('Filename.csv', usecols=[0, 1, 2, 3, 4, 5],
                 nrows=100)
```

It's possible to customize the headers, convert the columns or rows names and carry out a good number of other operations.

11.5.3 Importing data from Excel files

In the same way that we have worked with csv files, we can work with Excel file with the `read_excel` function, let's see some examples:
In this example, we want to load the sheet 1 from an Excel file:

```
import pandas as pd
df = pd.read_excel('Filename.xls', sheet_name='Sheet1')
```

This simple operation, loads the Sheet 1 from the Excel file into the Pandas DataFrame.

11.6 Indexing and Subsetting

Once we have the Pandas DataFrame prepared, independent of the source of our data (csv, Excel, hdf5, etc.) we can work with it, as if it were a database table, selecting the elements that interest us. We will work with some some examples on how to index and extract subsets of data.

Let's begin with loading a csv file having details of a market instrument.

```
In []: import pandas as pd
       df = pd.read_csv('MSFT.csv',
                        usecols = [0, 1, 2, 3, 4])
       df.head()
       df.shape

Out[]:     Date        Open    High    Low     Close
       0   2008-12-29  19.15   19.21   18.64   18.96
       1   2008-12-30  19.01   19.49   19.00   19.34
       2   2008-12-31  19.31   19.68   19.27   19.44
```

```
3   2009-01-02  19.53   20.40   19.37   20.33
4   2009-01-05  20.20   20.67   20.06   20.52

(1000, 5)
```

Here, we have read a csv file, of which we only need the columns of date, opening, closing, high and low (the first 5 columns) and we check the form of the DataFrame that has 1000 rows and 5 columns.

11.6.1 Selecting a single column

In the previous code, we have read directly the first 5 columns from the csv file. This is a filter that we applied, because we were only interested in those columns.

We can apply selection filters to the DataFrame itself, to select one column to work with. For example, we could need the `Close` column:

```
In []: close = df['Close']
       close.head()

Out[]:     Close
       0   18.96
       1   19.34
       2   19.44
       3   20.33
       4   20.52
```

11.6.2 Selecting multiple columns

We can select multiple columns too:

```
In []: closevol = df[['Close', 'Volume']]
       closevol.head()

Out[]:     Close   Volume
       0   18.96   58512800.0
       1   19.34   43224100.0
       2   19.44   46419000.0
```

```
3   20.33   50084000.0
4   20.52   61475200.0
```

11.6.3 Selecting rows via []

We can select a set of rows by index:

```
In []: import pandas as pd
       df = pd.read_csv('TSLA.csv' )
       df[100:110]
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

Or we can select a set of rows and columns:

```
In []: df[100:110][['Close', 'Volume']]

Out[]:
        Close   Volume
100 320.08  4236029.0
101 320.87  6942493.0
102 326.17  4980316.0
103 325.84  8547764.0
104 337.34  4463807.0
105 337.02  5715817.0
106 345.10  4888221.0
```

```
107 351.81  5032884.0
108 359.65  4898808.0
109 355.75  3280670.0
```

11.6.4 Selecting via `.loc[]` (By label)

With `df.loc` we can do the same selections using labels:
To select a set of rows, we can code the following using the index number as label:

```
In []: df.loc[100:110]
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

Or we can select a set of rows and columns like before:

```
In []: df.loc[100:110, ['Close', 'Volume']]
Out[]:
        Close   Volume
100 320.08  4236029.0
101 320.87  6942493.0
102 326.17  4980316.0
103 325.84  8547764.0
104 337.34  4463807.0
105 337.02  5715817.0
106 345.10  4888221.0
```

```
107 351.81  5032884.0
108 359.65  4898808.0
109 355.75  3280670.0
110 350.60  5353262.0
```

11.6.5 Selecting via `.iloc[]` (By position)

With `df.iloc` we can do the same selections using integer position:

```
In []: df.iloc[100:110]
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

In the last example, we used the index as an integer position rather than by label.
We can select a set of rows and columns like before:

```
In []: df.iloc[100:110, [3, 4]]
Out[]:
           Low     Close
100 317.25  320.08
101 316.66  320.87
102 323.20  326.17
103 323.56  325.84
104 336.16  337.34
105 336.25  337.02
```

```
106 344.34  345.10
107 348.20  351.81
108 354.13  359.65
109 350.07  355.75
```

11.6.6 Boolean indexing

So far, we have sliced subsets of data by label or by position. Now let's see how to select data that meet some criteria. We do this with Boolean indexing. We can use the same criteria similar to what we have seen with Numpy arrays. We show you just two illustrative examples here. This is by no means enough to get comfortable with it and so would encourage you to check the documentation and further readings at the end of this chapter to learn more.

- We can filter data that is greater (less) than a number.

```
In []: df[df.Close > 110]
Out[]:
```

	Date	Open	...	Close	...	Name
0	2017-03-27	304.00	...	279.18	...	TSLA
1	2017-03-26	307.34	...	304.18	...	TSLA
2	2017-03-23	311.25	...	301.54	...	TSLA
	...	...	...	...	...	...
1080	2017-12-09	137.00	...	141.60	...	TSLA
1081	2017-12-06	141.51	...	137.36	...	TSLA
1082	2013-12-05	140.51	...	140.48	...	TSLA

```
1083 rows × 14 columns

In []: df[(df['Close'] > 110) | (df['Close'] < 120)]
Out[]:
```

	Date	Open	...	Close	...	Name
0	2017-03-27	304.00	...	279.18	...	TSLA

	Date	Open	...	Close	...	Name
1	2017-03-26	307.34	...	304.18	...	TSLA
2	2017-03-23	311.25	...	301.54	...	TSLA
	...	...	...	...	...	...
1080	2017-12-09	137.00	...	141.60	...	TSLA
1081	2017-12-06	141.51	...	137.36	...	TSLA
1082	2013-12-05	140.51	...	140.48	...	TSLA

```
1083 rows × 14 columns
```

11.7 Manipulating a DataFrame

When we are working with data, the most common structure is the DataFrame. Until now we have seen how to create them, make selections and find data. We are now going to see how to manipulate the DataFrame to transform it into another DataFrame that has the form that our problem requires.

We'll see how to sort it, re-index it, eliminate unwanted (or spurious) data, add or remove columns and update values.

11.7.1 Transpose using `.T`

The Pandas DataFrame transpose function `T` allows us to transpose the rows as columns, and logically the columns as rows:

```
In []: import pandas as pd
       df = pd.read_csv('TSLA.csv' )
       df2 = df[100:110][['Close', 'Volume']]
       print(df2.T)
Out[]:
```

	100	101	102	...	109
Close	320.08	320.87	326.17	...	355.75
Volume	4236029.00	6942493.00	4980316.00	...	3280670.00

11.7.2 The `.sort_index()` method

When we are working with Pandas Dataframe it is usual to add or remove rows, order by columns, etc. That's why it's important to have a function that allows us to easily and comfortably sort the DataFrame by its index. We do this with the `sort_index` function of Pandas DataFrame.

```
In []: df.sort_index()
Out[]:
```

	Date	Open	High	Low	Close	...	Name
0	2017-03-27	304.00	304.27	277.18	279.18	...	TSLA
1	2017-03-26	307.34	307.59	291.36	304.18	...	TSLA
2	2017-03-23	311.25	311.61	300.45	301.54	...	TSLA
	...	...	...	...	...	...	...

11.7.3 The `.sort_values()` method

Sometimes, we may be interested in sorting the DataFrame by some column or even with several columns as criteria. For example, sort the column by first names and the second criterion by last names. We do this with the `sort_values` function of Pandas DataFrame.

```
In []: df.sort_values(by='Close')
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1081	2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
1057	2014-01-13	145.78	147.00	137.82	139.34	...	TSLA
1078	2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
	...	...	...	...	...	...	...

```
In []: df.sort_values(by=['Open', 'Close'])
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1080	2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
1077	2014-12-12	139.70	148.24	138.53	147.47	...	TSLA
1079	2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
	...	...	...	...	...	...	...

11.7.4 The `.reindex()` function

The Pandas' reindex funtion let us to realign the index of the Series or DataFrame, it's useful when we need to reorganize the index to meet some criteria. For example, we can play with the Series or DataFrame that we create before to alter the original index. For example, when the index is a label, we can reorganize as we need:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5),
                         index=['a', 'b', 'c', 'd', 'e'])
       df

Out[]:
```

	0
a	-0.134133
b	-0.586051
c	1.179358
d	0.433142
e	-0.365686

Now, we can reorganize the index as follows:

```
In []: df.reindex(['b', 'a', 'd', 'c', 'e'])
Out[]:
```

	0
b	-0.586051
a	-0.134133
d	0.433142
c	1.179358
e	-0.365686

When the index is numeric we can use the same function to order by hand the index:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5))
       df.reindex([4, 3, 2, 1, 0])
Out[]:
```

	0
4	1.058589
3	1.194400
2	-0.645806
1	0.836606
0	1.288102

Later in this section, we'll see how to work and reorganize date and time indices.

11.7.5 Adding a new column

Another interesting feature of DataFrames is the possibility of adding new columns to an existing DataFrame.
For example, we can add a new column to the random DataFrame that we have created before:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5))
Out[]:
```

	0
0	0.238304
1	2.068558
2	1.015650
3	0.506208
4	0.214760

To add a new column, we only need to include the new column name in the DataFrame and assign a initialization value, or assign to the new column a Pandas Series or another column from other DataFrame.

```
In []: df['new'] = 1
       print(df)
Out[]:
```

	0	new
0	0.238304	1
1	2.068558	1
2	1.015650	1
3	0.506208	1
4	0.214760	1

11.7.6 Delete an existing column

Likewise, we can remove one or more columns from the DataFrame. Let's create a DataFrame with 5 rows and 4 columns with random values to delete one column.

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4))
       print(df)
Out[]:
```

	0	1	2	3
0	-1.171562	-0.086348	-1.971855	1.168017
1	-0.408317	-0.061397	-0.542212	-1.412755
2	-0.365539	-0.587147	1.494690	1.756105
3	0.642882	0.924202	0.517975	-0.914366
4	0.777869	-0.431151	-0.401093	0.145646

Now, we can delete the column that we specify by index or by label if any:

```
In []: del df[0]
Out[]:
```

	1	2	3
0	-0.086348	-1.971855	1.168017
1	-0.061397	-0.542212	-1.412755
2	-0.587147	1.494690	1.756105
3	0.924202	0.517975	-0.914366
4	-0.431151	-0.401093	0.145646

```
In []: df['new'] = 1
       print(df)
Out[]:
```

	0	new
0	0.238304	1
1	2.068558	1
2	1.015650	1
3	0.506208	1

```
In []: del df['new']
Out[]:
```

	0
0	0.238304
1	2.068558
2	1.015650
3	0.506208

11.7.7 The `.at[]` (By label)

Using `at`, we can to locate a specific value by row and column labels as follows:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5,4),
                         index=['a', 'b', 'c', 'd', 'e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	0.996496	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

```
In []: df.at['a', 'A']
Out[]: 0.9964957014209125
```

It is possible to assign a new value with the same funcion too:

```
In []: df.at['a', 'A'] = 0
Out[]:
```

	A	B	C	D
a	0.000000	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

11.7.8 The `.iat[]` (By position)

With `iat` we can to locate a specific value by row and column index as follow:

```
In []: import pandas as pd
              import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a', 'b', 'c', 'd', 'e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
       df.iat[0, 0]

Out[]: 0.996496
```

It is possible to assign a new value with the same funcion too:

```
In []: df.iat[0, 0] = 0
Out[]:
```

	A	B	C	D
a	0.000000	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

11.7.9 Conditional updating of values

Another useful function is to update values that meet some criteria, for example, update values whose values are greater than 0:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)

       df[df > 0] = 1
       print(df)
```

Out[]:

	A	B	C	D
a	1.000000	-0.082466	1.000000	-0.728372
b	-0.784404	-0.663096	-0.595112	1.000000
c	-1.460702	-1.072931	-0.761314	1.000000
d	1.000000	1.000000	1.000000	-0.302310
e	-0.488556	1.000000	-0.798716	-0.590920

We can also update the values of a specific column that meet some criteria, or even work with several columns as criteria and update a specific column.

```
In []: df['A'][df['A'] < 0] = 1
       print(df)
```

Out[]:

	A	B	C	D
a	1.0	-0.082466	1.000000	-0.728372
b	1.0	-0.663096	-0.595112	1.000000
c	1.0	-1.072931	-0.761314	1.000000

	A	B	C	D
d	1.0	1.000000	1.000000	-0.302310
e	1.0	1.000000	-0.798716	-0.590920

```
In []: df['A'][(df['B'] < 0 ) & (df['C'] < 0)] = 9
       print(df)
Out[]:
```

	A	B	C	D
a	1.0	-0.082466	1.000000	-0.728372
b	9.0	-0.663096	-0.595112	1.000000
c	9.0	-1.072931	-0.761314	1.000000
d	1.0	1.000000	1.000000	-0.302310
e	1.0	1.000000	-0.798716	-0.590920

11.7.10 The `.dropna()` method

Occasionally, we may have a DataFrame that, for whatever reason, includes NA values. This type of values is usually problematic when we are making calculations or operations and must be treated properly before proceeding with them. The easiest way to eliminate NA values is to remove the row that contains it.

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)

Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332

	A	B	C	D
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	0.312319	-0.765930	-1.641234	-1.388924

```
In []: df['A'][(df['B'] < 0 ) & (df['C'] < 0)] = np.nan
       print(df)
```

```
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	NaN	-0.765930	-1.641234	-1.388924

```
In []: df = df.dropna()
        print(df)
```

```
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177

Here we are deleting the whole row that has, in any of its columns, a NaN value, but we can also specify that it deletes the column that any of its values is NaN:

```
df = df.dropna(axis=1)
print(df)
```

We can specify if a single NaN value is enough to delete the row or column, or if the whole row or column must have NaN to delete it.

```
df = df.dropna(how='all')
print(df)
```

11.7.11 The `.fillna()` method

With the previous function we have seen how to eliminate a complete row or column that contains one or all the values to NaN, this operation can be a little drastic if we have valid values in the row or column.

For this, it is interesting to use the `fillna` function that substitutes the NaN values with some fixed value.

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	0.312319	-0.765930	-1.641234	-1.388924

```
In []: df['A'][(df['B'] < 0 ) & (df['C'] < 0)] = np.nan
       print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889

	A	B	C	D
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	NaN	-0.765930	-1.641234	-1.388924

```
In []: df = df.fillna(999)
       print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	999	-0.765930	-1.641234	-1.388924

11.7.12 The `.apply()` method

The `apply` is a very useful way to use functions or methods in a DataFrame without having to loop through it. We can apply the `apply` method to a Series or DataFrame to apply a function to all rows or columns of the DataFrame. Let's see some examples.

Suppose we are working with the randomly generated DataFrame and need to apply a function. In this example, for simplicity's sake, we're going to create a custom function to square a number.

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
def square_number(number):
  return number**2

# Test the function
In []: square_number(2)
Out[]: 4
```

Now, let's use the custom function through Apply:

```
In []: df.apply(square_number, axis=1)
Out[]:
```

	A	B	C	D
a	0.401005	7.285074	0.329536	0.426073
b	0.003636	0.022658	0.022238	0.491704
c	0.002758	0.220412	0.808524	0.370161
d	1.830372	0.010671	0.209652	3.599253
e	0.007793	0.174989	1.216586	0.339254

This method `apply` the funcion `square_number` to all rows of the DataFrame.

11.7.13 The `.shift()` function

The `shift` function allows us to move a row to the right or left and/or to move a column up or down. Let's look at some examples.

First, we are going to move the values of a column downwards:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df['D'] = df['D'].shift(1)
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	NaN
b	0.060295	-0.150527	0.149123	0.652742
c	-0.052515	0.469481	0.899180	-0.701216
d	-1.352912	0.103302	0.457878	-0.608409
e	0.088279	0.418317	-1.102989	-1.897170

We are going to move the values of a column upwards

```
In []: df['D'] = df['D'].shift(-1)
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	-0.701216

	A	B	C	D
b	0.060295	-0.150527	0.149123	-0.608409
c	-0.052515	0.469481	0.899180	-1.897170
d	-1.352912	0.103302	0.457878	0.582455
e	0.088279	0.418317	-1.102989	NaN

This is very useful for comparing the current value with the previous value.

11.8 Statistical Exploratory data analysis

Pandas DataFrame allows us to make some descriptive statistics calculations, which are very useful to make a first analysis of the data we are handling. Let's see some useful functions.

11.8.1 The `info()` function

It is a good practice to know the structure and format of our DataFrame, the `Info` function offers us just that:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df.info()
```

```
Out[]: <class 'pandas.core.frame.DataFrame'>
       Index: 5 entries, a to e
       Data columns (total 5 columns):
       A         5 non-null float64
       B         5 non-null float64
       C         5 non-null float64
       D         5 non-null float64
       shift     4 non-null float64
       dtypes: float64(5)
       memory usage: 240.0+ bytes
```

11.8.2 The `describe()` function

We can obtain a statistical overview of the DataFrame with the 'describe function, which gives us the mean, median, standard deviation, maximum, minimum, quartiles, etc. of each DataFrame column.

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df.describe()
Out[]:
```

	A	B	C	D
count	5.000000	5.000000	5.000000	5.000000
mean	-0.378020	-0.371703	0.195449	-0.394319
std	0.618681	1.325046	0.773876	1.054633
min	-1.352912	-2.699088	-1.102989	-1.897170
25%	-0.633249	-0.150527	0.149123	-0.701216
50%	-0.052515	0.103302	0.457878	-0.608409
75%	0.060295	0.418317	0.574052	0.582455
max	0.088279	0.469481	0.899180	0.652742

11.8.3 The `value_counts()` function

The function `value_counts` counts the repeated values of the specified column:

```
In []: df['A'].value_counts()
Out[]: 0.088279    1
      -0.052515    1
       0.060295    1
      -0.633249    1
      -1.352912    1
      Name: A, dtype: int64
```

11.8.4 The `mean()` function

We can obtain the mean of a specific column or row by means of the `mean` function.

```
In []: df['A'].mean() # Specifying a column
Out[]: -0.3780203497252693

In []: df.mean() # By column
       df.mean(axis=0) # By column

Out[]:  A       -0.378020
        B       -0.371703
        C        0.195449
        D       -0.394319
```

```
shift   -0.638513
dtype: float64

In []: df.mean(axis=1) # By row

Out[]:  a   -0.526386
        b    0.002084
        c    0.001304
        d   -0.659462
        e   -0.382222
        dtype: float64
```

11.8.5 The `std()` function

We can obtain the standard deviation of a specific column or row by means of the `std` function.

```
In []: df['A'].std() # Specifying a column
Out[]: 0.6186812554819784

In []: df.std() # By column
       df.std(axis=0) # By column

Out[]:  A        0.618681
        B        1.325046
        C        0.773876
        D        1.054633
        shift    1.041857
        dtype: float64

In []: df.std(axis=1) # By row

Out[]:  a    1.563475
        b    0.491499
        c    0.688032
        d    0.980517
        e    1.073244
        dtype: float64
```

11.9 Filtering Pandas DataFrame

We have already seen how to filter data in a DataFrame, including logical statements to filter rows or columns with some logical criteria. For example, we will filter rows whose column 'A' is greater than zero:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4),
                         index=['a','b','c','d','e'],
                         columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df_filtered = df[df['A'] > 0]
       print(df_filtered)
Out[]:
```

	A	B	C	D
b	0.060295	-0.150527	0.149123	-0.701216
e	0.088279	0.418317	-1.102989	0.582455

We can also combine logical statements, we will filter all rows whose column 'A' and 'B' have their values greater than zero.

```
In []: df_filtered = df[(df['A'] > 0) & (df['B'] > 0)]
       print(df_filtered)
Out[]:
```

	A	B	C	D
e	0.088279	0.418317	-1.102989	0.582455

11.10 Iterating Pandas DataFrame

We can go through the DataFrame row by row to do operations in each iteration, let's see some examples.

```
In []: for item in df.iterrows():
           print(item)
Out[]:

('a', A       -0.633249
B       -2.699088
C        0.574052
D        0.652742
shift         NaN
Name: a, dtype: float64)
('b', A        0.060295
B       -0.150527
C        0.149123
D       -0.701216
shift    0.652742
Name: b, dtype: float64)
('c', A       -0.052515
B        0.469481
C        0.899180
D       -0.608409
shift   -0.701216
Name: c, dtype: float64)
('d', A       -1.352912
B        0.103302
C        0.457878
D       -1.897170
shift   -0.608409
Name: d, dtype: float64)
('e', A        0.088279
```

```
B         0.418317
C        -1.102989
D         0.582455
shift    -1.897170
Name: e, dtype: float64)
```

11.11 Merge, Append and Concat Pandas DataFrame

Another interesting feature of DataFrames is that we can merge, concatenate them and add new values, let's see how to do each of these operations.

- `merge` function allows us to merge two DataFrame by rows:

```
In []: import pandas as pd
       import numpy as np
       df1 = pd.DataFrame(np.random.randn(5, 4),
                          index = ['a','b','c','d','e'],
                          columns = ['A', 'B', 'C', 'D'])
       print(df1)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334

```
In []: import pandas as pd
       import numpy as np
       df2 = pd.DataFrame(np.random.randn(5, 4),
                          index = ['a','b','c','d','e'],
                          columns = ['A', 'B', 'C', 'D'])
       print(df2)
Out[]:
```

	A	B	C	D
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
In []: df3 = pd.merge(df1, df2)
       print(df3)

Out[]: Empty DataFrame
       Columns: [A, B, C, D]
       Index: []
```

- `append` function allows us to append rows from one DataFrame to another DataFrame by rows:

```
In []: import pandas as pd
       import numpy as np
       df1 = pd.DataFrame(np.random.randn(5, 4),
                          index=['a', 'b', 'c', 'd', 'e'],
                          columns=['A', 'B', 'C', 'D'])

In []: df2 = pd.DataFrame(np.random.randn(5, 4),
                          index=['a', 'b', 'c', 'd', 'e'],
                          columns=['A', 'B', 'C', 'D'])

In []: df3 = df1.append(df2)
       print(df3)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334

	A	B	C	D
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

- `concat` function allows us to merge two DataFrame by rows or columns:

```
In []: import pandas as pd
       import numpy as np
       df1 = pd.DataFrame(np.random.randn(5, 4),
                          index=['a', 'b', 'c', 'd', 'e'],
                          columns=['A', 'B', 'C', 'D'])

In []: df2 = pd.DataFrame(np.random.randn(5, 4),
                          index=['a', 'b', 'c', 'd', 'e'],
                          columns=['A', 'B', 'C', 'D'])

In []: df3 = pd.concat([df1, df2]) # Concat by row
       print(df3)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
In []: df3 = pd.concat([df1, df2], axis=0) # Concat by row
       print(df3)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
# Concat by column
In []: df3 = pd.concat([df1, df2], axis=1)
       print(df3)
Out[]:
```

	A	B	...	D	A	...	D
a	1.179924	-1.512124	...	0.019265	2.030462	...	-0.757323
b	0.019969	-1.351649	...	-0.989025	0.475807	...	-0.843963
c	0.351921	-0.792914	...	0.170751	0.948164	...	1.319999
d	-0.150499	0.151942	...	-0.347300	1.433736	...	-1.051454
e	-1.307590	0.185759	...	-0.170334	0.565345	...	-0.227519

11.12 TimeSeries in Pandas

Pandas TimeSeries includes a set of tools to work with Series or DataFrames indexed in time. Usually, the series of financial data are of this type and therefore, knowing these tools will make our work much more comfortable. We are going to start creating time series from scratch and then we will see

how to manipulate them and convert them to different frequencies.

11.12.1 Indexing Pandas TimeSeries

With `date_range` Panda's method, we can create a time range with a certain frequency. For example, create a range starting in December 1st, 2018, with 30 occurrences with an hourly frequency.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
                          freq='H')
       print(rng)

Out[]: DatetimeIndex([
       '2018-12-01 00:00:00', '2018-12-01 01:00:00',
       '2018-12-01 02:00:00', '2018-12-01 03:00:00',
       '2018-12-01 04:00:00', '2018-12-01 05:00:00',
       '2018-12-01 06:00:00', '2018-12-01 07:00:00',
       '2018-12-01 08:00:00', '2018-12-01 09:00:00',
       '2018-12-01 10:00:00', '2018-12-01 11:00:00',
       '2018-12-01 12:00:00', '2018-12-01 13:00:00',
       '2018-12-01 14:00:00', '2018-12-01 15:00:00',
       '2018-12-01 16:00:00', '2018-12-01 17:00:00',
       '2018-12-01 18:00:00', '2018-12-01 19:00:00',
       '2018-12-01 20:00:00', '2018-12-01 21:00:00',
       '2018-12-01 22:00:00', '2018-12-01 23:00:00',
       '2018-12-02 00:00:00', '2018-12-02 01:00:00',
       '2018-12-02 02:00:00', '2018-12-02 03:00:00',
       '2018-12-02 04:00:00', '2018-12-02 05:00:00'],
       dtype='datetime64[ns]', freq='H')
```

We can do the same to get a daily frequency[2] (or any other, as per our requirement). We can use the `freq` parameter to adjust this.

```
In []: rng = pd.date_range('12/1/2018', periods=10,
                          freq='D')
       print(rng)
```

[2]https://pandas.pydata.org/pandas-docs/stable/reference/api/pandasrange.html

```
Out[]: DatetimeIndex(['2018-12-01', '2018-12-02',
         '2018-12-03', '2018-12-04', '2018-12-05',
         '2018-12-06', '2018-12-07', '2018-12-08',
         '2018-12-09', '2018-12-10'],
         dtype='datetime64[ns]', freq='D')
```

Now, we have a `DateTimeIndex` in the `rng` object and we can use it to create a Series or DataFrame:

```
In []: ts = pd.DataFrame(np.random.randn(len(rng), 4),
       index=rng, columns=['A', 'B', 'C', 'D'])
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.048603	0.968522	0.408213	0.921774
2018-12-02	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-03	-2.337844	0.329954	0.289221	0.259132
2018-12-04	1.357521	0.969808	1.341875	0.767797
2018-12-05	-1.212355	-0.077457	-0.529564	0.375572
2018-12-06	-0.673065	0.527754	0.006344	-0.533316
2018-12-07	0.226145	0.235027	0.945678	-1.766167
2018-12-08	1.735185	-0.604229	0.274809	0.841128

```
In []: ts = pd.Series(np.random.randn(len(rng)), index=rng)
       print(ts)

Out[]: 2018-12-01    0.349234
       2018-12-02   -1.807753
       2018-12-03    0.112777
       2018-12-04    0.421516
       2018-12-05   -0.992449
       2018-12-06    1.254999
       2018-12-07   -0.311152
       2018-12-08    0.331584
       2018-12-09    0.196904
       2018-12-10   -1.619186
```

```
2018-12-11    0.478510
2018-12-12   -1.036074
```

Sometimes, we read the data from internet sources or from csv files and we need to convert the date column into the index to work properly with the Series or DataFrame.

```
In []: import pandas as pd
       df = pd.read_csv('TSLA.csv')
       df.tail()
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1078	2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
1079	2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
1080	2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
1081	2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
1082	2013-12-05	140.15	143.35	139.50	140.48	...	TSLA

Here, we can see the index as numeric and a Date column, let's convert this column into the index to indexing our DataFrame, read from a csv file, in time. For this, we are going to use the Pandas `set_index` method

```
In []: df = df.set_index('Date')
       df.tail()
Out[]:
```

Date	Open	High	Low	Close	...	Name
2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
2013-12-05	140.15	143.35	139.50	140.48	...	TSLA

Now, we have Pandas TimeSeries ready to work.

11.12.2 Resampling Pandas TimeSeries

A very useful feature of Pandas TimeSeries is the resample capacity, this allows us to pass the current frequency to another higher frequency (we can't pass to lower frequencies, because we don't know the data).

As it can be supposed, when we pass from one frequency to another data could be lost, for this, we must use some function that treat the values of each frequency interval, for example, if we pass from an hourly frequency to daily, we must specify what we want to do with the group of data that fall inside each frequency, we can do a mean, a sum, we can get the maximum or the minimum, etc.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
              freq='H')
       ts = pd.DataFrame(np.random.randn(len(rng), 4),
            index=rng, columns=['A', 'B', 'C', 'D'])
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01 00:00:00	0.048603	0.968522	0.408213	0.921774
2018-12-01 01:00:00	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-01 02:00:00	-2.337844	0.329954	0.289221	0.259132
2018-12-01 03:00:00	1.357521	0.969808	1.341875	0.767797
2018-12-01 04:00:00	-1.212355	-0.077457	-0.529564	0.375572
2018-12-01 05:00:00	-0.673065	0.527754	0.006344	-0.533316

```
In []: ts = ts.resample("1D").mean()
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.449050	0.127412	-0.154179	-0.358324
2018-12-02	-0.539007	-0.855894	0.000010	0.454623

11.12.3 Manipulating TimeSeries

We can manipulate the Pandas TimeSeries in the same way that we have done until now, since they offer us the same capacity that the Pandas Series and the Pandas DataFrames. Additionally, we can work comfortably with all jobs related to handling dates. For example, to obtain all the data from a date, to obtain the data in a range of dates, etc.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
              freq='D')
       ts = pd.DataFrame(np.random.randn(len(rng), 4),
            index=rng, columns=['A', 'B', 'C', 'D'])
       print(ts.head())
Out[]:
```

	A	B	C	D
2018-12-01	0.048603	0.968522	0.408213	0.921774
2018-12-02	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-03	-2.337844	0.329954	0.289221	0.259132
2018-12-04	1.357521	0.969808	1.341875	0.767797
2018-12-05	-1.212355	-0.077457	-0.529564	0.375572

Getting all values from a specific date:

```
In []: ts['2018-12-15':].head()
Out[]:
```

	A	B	C	D
2018-12-15	0.605576	0.584369	-1.520749	-0.242630
2018-12-16	-0.105561	-0.092124	0.385085	0.918222
2018-12-17	0.337416	-1.367549	0.738320	2.413522
2018-12-18	-0.011610	-0.339228	-0.218382	-0.070349
2018-12-19	0.027808	-0.422975	-0.622777	0.730926

Getting all values inside a date range:

```
In []: ts['2018-12-15':'2018-12-20']
Out[]:
```

	A	B	C	D
2018-12-15	0.605576	0.584369	-1.520749	-0.242630
2018-12-16	-0.105561	-0.092124	0.385085	0.918222
2018-12-17	0.337416	-1.367549	0.738320	2.413522
2018-12-18	-0.011610	-0.339228	-0.218382	-0.070349
2018-12-19	0.027808	-0.422975	-0.622777	0.730926
2018-12-20	0.188822	-1.016637	0.470874	0.674052

11.13 Key Takeaways

1. Pandas DataFrame and Pandas Series are some of the most important data structures. It is a must to acquire fluency in its handling because we will find them in practically all the problems that we handle.
2. A DataFrame is a data structure formed by rows and columns and has an index.
3. We must think of them as if they were data tables (for the Array with a single column) with which we can select, filter, sort, add and delete elements, either by rows or columns.
4. Help in ETL processes (Extraction, Transformation and Loading)
5. We can select, insert, delete and update elements with simple functions.
6. We can perform computations by rows or columns.
7. Has the ability to run vectorized computations.
8. We can work with several DataFrames at the same time.
9. Indexing and subsetting are the most important features from Pandas.
10. Facilitates the statistical exploration of the data.
11. It offers us a variety of options for handling NaN data.
12. Another additional advantage is the ability to read & write multiple data formats (CSV, Excel, HDF5, etc.).
13. Retrieve data from external sources (Yahoo, Google, Quandl, etc.)
14. Finally, it has the ability to work with date and time indexes and offers us a set of functions to work with dates.

Chapter 12

Data Visualization with Matplotlib

Matplotlib is a popular Python library that can be used to create data visualizations quite easily. It is probably the single most used Python package for 2D-graphics along with limited support for 3D-graphics. It provides both, a very quick way to visualize data from Python and publication-quality figures in many formats. Also, It was designed from the beginning to serve two purposes:

1. Allow for interactive, cross-platform control of figures and plots
2. Make it easy to produce static vector graphics files without the need for any GUIs.

Much like Python itself, Matplotlib gives the developer complete control over the appearance of their plots. It tries to make easy things easy and hard things possible. We can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc. with just a few lines of code. For simple plotting, the `pyplot` module within `matplotlib` package provides a MATLAB-like interface to the underlying object-oriented plotting library. It implicitly and automatically creates figures and axes to achieve the desired plot.

To get started with Matplotlib, we first *import* the package. It is a common practice to import `matplotlib.pyplot` using the alias as `plt`. The `pyplot` being the sub-package within Matplotlib provides the common charting

functionality. Also, if we are working in a Jupyter Notebook, the line `%matplotlib inline` becomes important, as it makes sure that the plots are embedded inside the notebook. This is demonstrated in the example below:

```
import matplotlib.pyplot as plt

%matplotlib inline
```

NOTE: `Matplotlib` does not fall under the Python Standard Library and hence, like any other third party library, it needs to be installed before it can be used. It can be installed using the command `pip install matplotlib`.

12.1 Basic Concepts

Matplotlib allows creating a wide variety of plots and graphs. It is a humongous project and can seem daunting at first. However, we will break it down into bite-sized components and so learning it should be easier.
Different sources use 'plot' to mean different things. So let us begin by defining specific terminology used across the domain.

- `Figure` is the top-level container in the hierarchy. It is the overall window where everything is drawn. We can have multiple independent figures, and each figure can have multiple `Axes`. It can be created using the `figure` method of `pyplot` module.
- `Axes` is where the plotting occurs. The axes are effectively the area that we plot data on. Each `Axes` has an `X-Axis` and a `Y-Axis`.

The below mentioned example illustrates the use of the above-mentioned terms:

```
fig = plt.figure()
<Figure size 432x288 with 0 Axes>
```

Upon running the above example, nothing happens really. It only creates a figure of size 432 x 288 with 0 Axes. Also, Matplotlib will not show anything until told to do so. Python will wait for a call to `show` method to display the

plot. This is because we might want to add some extra features to the plot before displaying it, such as title and label customization. Hence, we need to call `plt.show()` method to show the figure as shown below:

```
plt.show()
```

As there is nothing to plot, there will be no output. While we are on the topic, we can control the size of the figure through the `figsize` argument, which expects a tuple of (width, height) in inches.

```
fig = plt.figure(figsize=(8, 4))
<Figure size 576x288 with 0 Axes>
plt.show()
```

12.1.1 Axes

All plotting happens with respect to an `Axes`. An Axes is made up of `Axis` objects and many other things. An *Axes* object must belong to a `Figure`. Most commands that we will ever issue will be with respect to this *Axes* object. Typically, we will set up a `Figure`, and then add `Axes` on to it. We can use `fig.add_axes` but in most cases, we find that adding a subplot fits our need perfectly. A subplot is an axes on a grid system.

- `add_subplot` method adds an `Axes` to the figure as part of a subplot arrangement.

```
# -Example 1-
# Creating figure
fig = plt.figure()

# Creating subplot
# Sub plot with 1 row and 1 column at the index 1
ax = fig.add_subplot(111)
plt.show()
```

The above code adds a single plot to the figure `fig` with the help of `add_subplot()` method. The output we get is a blank plot with axes ranging from 0 to 1 as shown in *figure 1*.

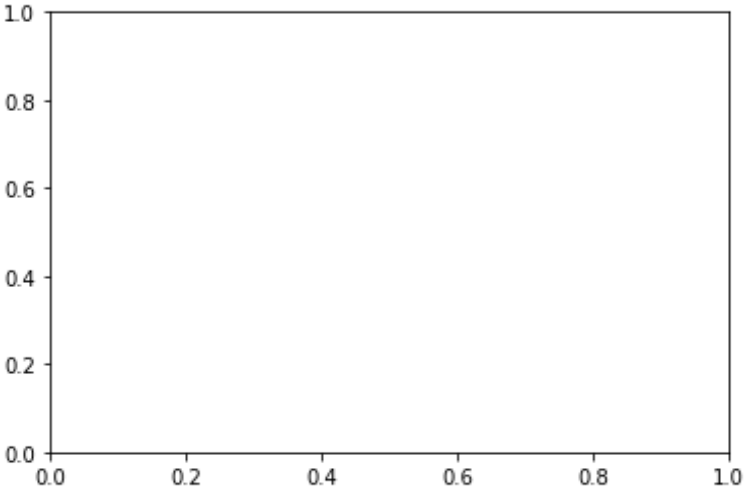

Figure 1: Empty plot added on axes

We can customize the plot using a few more built-in methods. Let us add the title, X-axis label, Y-axis label, and set limit range on both axes. This is illustrated in the below code snippet.

```
# -Example 2-
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting axes/plot title
ax.set_title('An Axes Title')

# Setting X-axis and Y-axis limits
ax.set_xlim([0.5, 4.5])
ax.set_ylim([-3, 7])

# Setting X-axis and Y-axis labels
ax.set_ylabel('Y-Axis Label')
ax.set_xlabel('X-Axis Label')
```

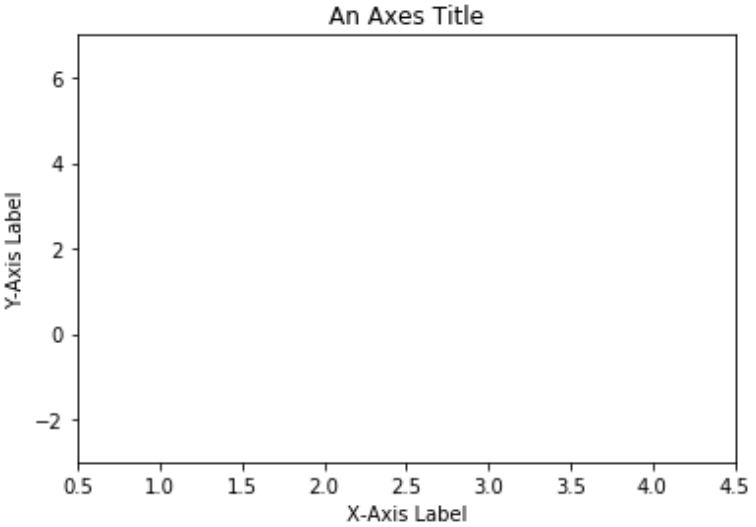

Figure 2: An empty plot with title, labels and custom axis limits

```
# Showing the plot
plt.show()
```

The output of the above code is shown in *figure 2*. Matplotlib's objects typically have lots of *explicit setters*, i.e. methods that start with `set_<something>` and control a particular option. Setting each option using explicit setters becomes repetitive, and hence we can set all required parameters directly on the axes using the `set` method as illustrated below:

```
# -Example 2 using the set method-
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting title and axes properties
ax.set(title='An Axes Title', xlim=[0.5, 4.5],
       ylim=[-3, 7], ylabel='Y-Axis Label',
       xlabel='X-Axis Label')

plt.show()
```

NOTE: The `set` method does not just apply to `Axes`; it applies to more-or-less all matplotlib objects.

The above code snippet gives the same output as *figure 2*. Using the `set` method when all required parameters are passed as arguments.

12.1.2 Axes method v/s `pyplot`

Interestingly, almost all methods of axes objects exist as a method in the `pyplot` module. For example, we can call `plt.xlabel('X-Axis Label')` to set label of X-axis (`plt` being an alias for `pyplot`), which in turn calls `ax.set_xlabel('X-Axis Label')` on whichever axes is *current*.

```
# -Example 3-
# Creating subplots, setting title and axes labels
# using `pyplot`
plt.subplots()
plt.title('Plot using pyplot')
plt.xlabel('X-Axis Label')
plt.ylabel('Y-Axis Label')
plt.show()
```

The code above is more intuitive and has fewer variables to construct a plot. The output for the same is shown in *figure 3*. It uses implicit calls to axes method for plotting. However, if we take a look at *"The Zen of Python"* (try `import this`), it says:

```
"Explicit is better than implicit."
```

While very simple plots, with short scripts, would benefit from the conciseness of the `pyplot` implicit approach, when doing more complicated plots, or working within larger scripts, we will want to explicitly pass around the axes and/or figure object to operate upon. We will be using both approaches here wherever it deems appropriate.

Anytime we see something like below:

```
fig = plt.figure()
ax = fig.add_subplot(111)
```

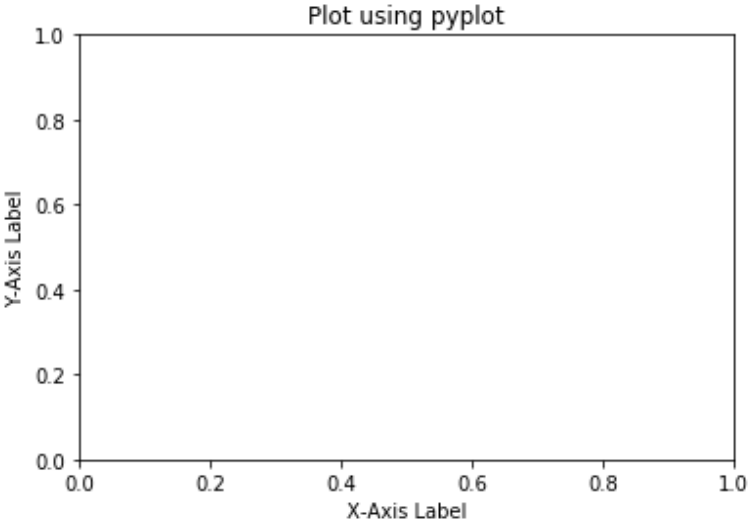

Figure 3: An empty plot using `pyplot`

can almost always be replaced with the following code:

```
fig, ax = plt.subplots()
```

Both versions of code produce the same output. However, the latter version is cleaner.

12.1.3 Multiple Axes

A figure can have more than one `Axes` on it. The easiest way is to use `plt.subplots()` call to create a figure and add the axes to it automatically. Axes will be on a regular grid system. For example,

```
# -Example 4-
# Creating subplots with 2 rows and 2 columns
fig, axes = plt.subplots(nrows=2, ncols=2)
plt.show()
```

Upon running the above code, Matplotlib would generate a figure with four subplots arranged with two rows and two columns as shown in *figure 4*.

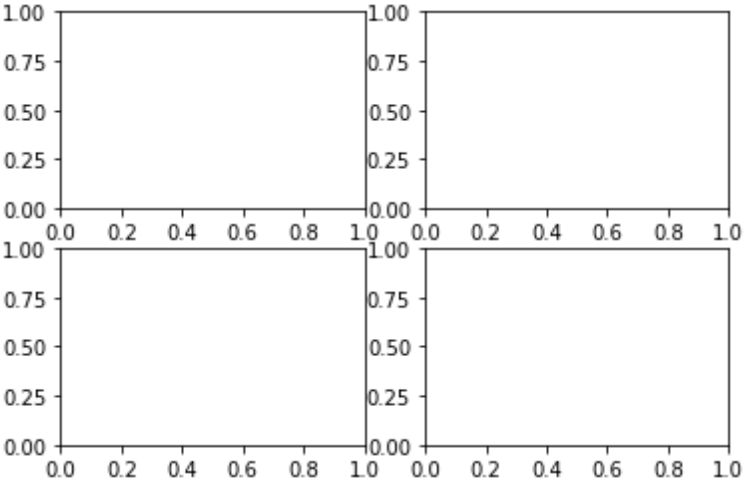

Figure 4: Figure with multiple axes

The `axes` object that was returned here would be a 2D-NumPy array, and each item in the array is one of the subplots. Therefore, when we want to work with one of these axes, we can index it and use that item's methods. Let us add the title to each subplot using the axes methods.

```
# -Example 5-
# Create a figure with four subplots and shared axes
fig, axes = plt.subplots(nrows=2, ncols=2, sharex=True,
sharey=True)
axes[0, 0].set(title='Upper Left')
axes[0, 1].set(title='Upper Right')
axes[1, 0].set(title='Lower Left')
axes[1, 1].set(title='Lower Right')
plt.show()
```

The above code generates a figure with four subplots and shared X and Y axes. Axes are shared among subplots in row wise and column-wise manner. We then set a title to each subplot using the `set` method for each subplot. Subplots are arranged in a clockwise fashion with each subplot having a unique index. The output is shown in *figure 5*.

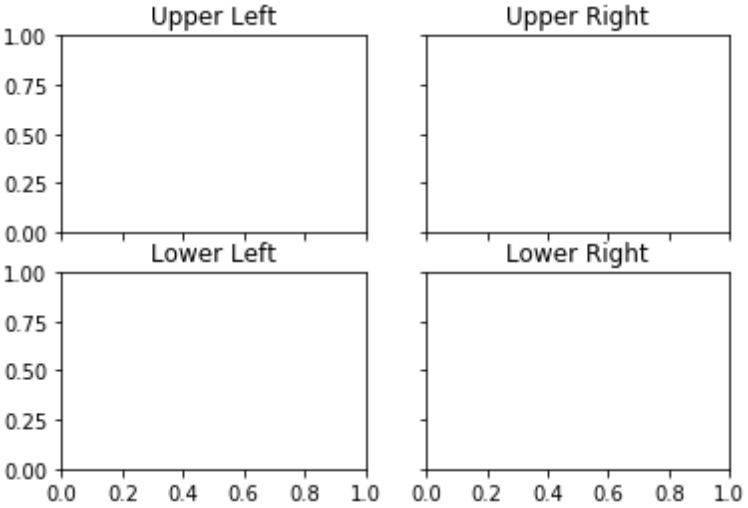

Figure 5: Subplots with the share axes

12.2 Plotting

We have discussed a lot about laying things out, but we haven't really discussed anything about plotting data yet. Matplotlib has various plotting functions. Many more than we will discuss and cover here. However, a full list or gallery[1] can be a bit overwhelming at first. Hence, we will condense it down and attempt to start with simpler plotting and than move towards more complex plotting. The `plot` method of `pyplot` is one of the most widely used methods in Matplotlib to plot the data. The syntax to call the `plot` method is shown below:

```
plot([x], y, [fmt], data=None, **kwargs)
```

The coordinates of the points or line nodes are given by x and y. The optional parameter *fmt* is a convenient way of defining basic formatting like color, market, and style. The `plot` method is used to plot almost any kind of data in Python. It tells Python what to plot and how to plot it, and also allows customization of the plot being generated such as color, type, etc.

[1] https://matplotlib.org/gallery/index.html

12.2.1 Line Plot

A line plot can be plotted using the `plot` method. It plots Y versus X as lines and/or markers. Below we discuss a few scenarios for plotting line. To plot a line, we provide coordinates to be plotted along X and Y axes separately as shown in the below code snippet.

```
# -Example 6-
# Defining coordinates to be plotted on X and Y axes
# respectively
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot lists 'x' and 'y'
plt.plot(x, y)

# Plot axes labels and show the plot
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.show()
```

The above code plots values in the list `x` along the X-axis and values in the list `y` along the Y-axis as shown in *figure 6*.

The call to `plot` takes minimal arguments possible, i.e. values for Y-axis only. In such a case, Matplotlib will implicitly consider the index of elements in list `y` as the input to the X-axis as demonstrated in the below example:

```
# -Example 7-
# Defining 'y' coordinates
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot list 'y'
plt.plot(y)

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

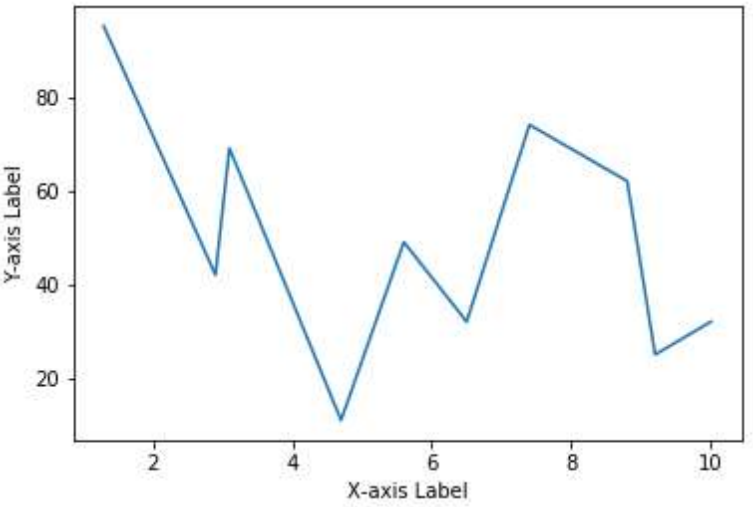

Figure 6: Line plot with x *and* y *as data*

Here, we define a list called `y` that contains values to be plotted on Y-axis. The output for the same is shown in *figure 7*.

The plots created uses *the default* line style and color. The optional parameter *fmt* in the `plot` method is a convenient way for defining basic formatting like color, marker, and line-style. It is a shortcut string notation consisting of color, marker, and line:

```
fmt = '[color][marker][line]'
```

Each of them is optional. If not provided, the value from the style cycle[2] is used. We use this notation in the below example to change the line color:

```
# -Example 8-
# Plot line with green color
plt.plot(y, 'g')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
```

[2]https://matplotlib.org/tutorials/intermediate/color_cycle.html

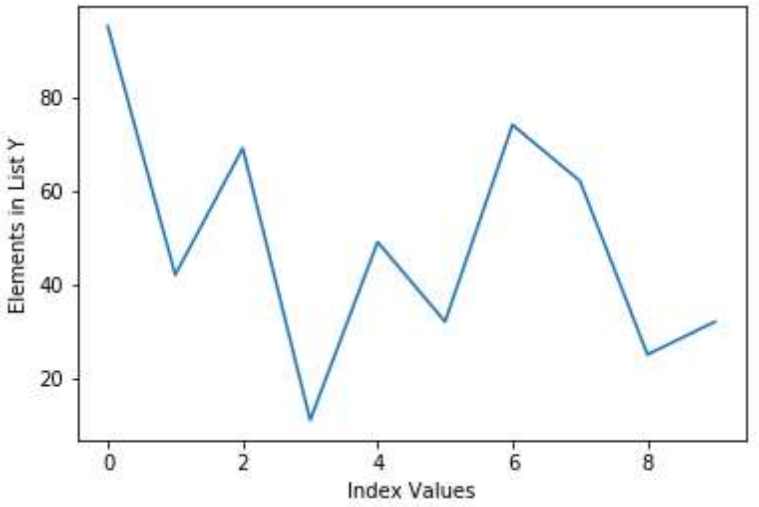

Figure 7: Line plot with `y` as the only data

```
plt.ylabel('Elements in List Y')
plt.show()
```

Following the *fmt* string notation, we changed the color of a line to green using the character `g` which refers to the line color. This generates the plot with green line as shown in *figure 8*. Likewise, markers are added using the same notation as shown below:

```
# -Example 9-
# Plot continuous green line with circle markers
plt.plot(y, 'go-')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

Here, the *fmt* parameters: `g` refers to the green color, `o` refers to circle markers and `-` refers to a continuous line to be plotted as shown in *figure 9*. This formatting technique allows us to format a line plot in virtually any way

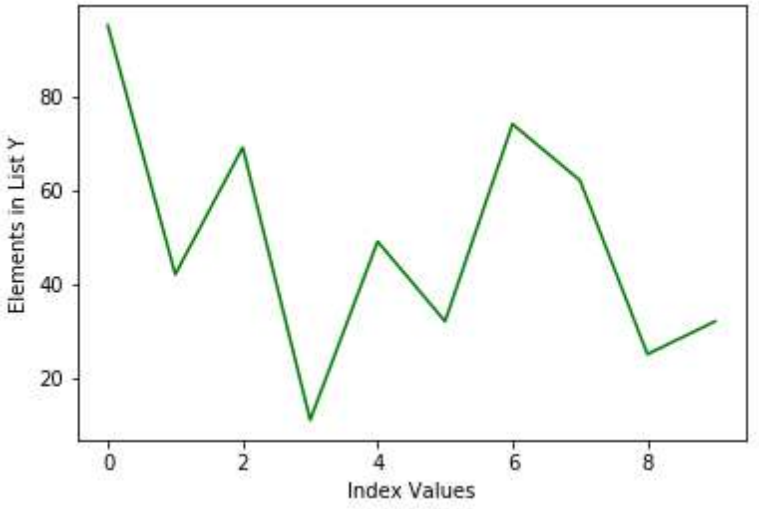

Figure 8: Line plot with green line

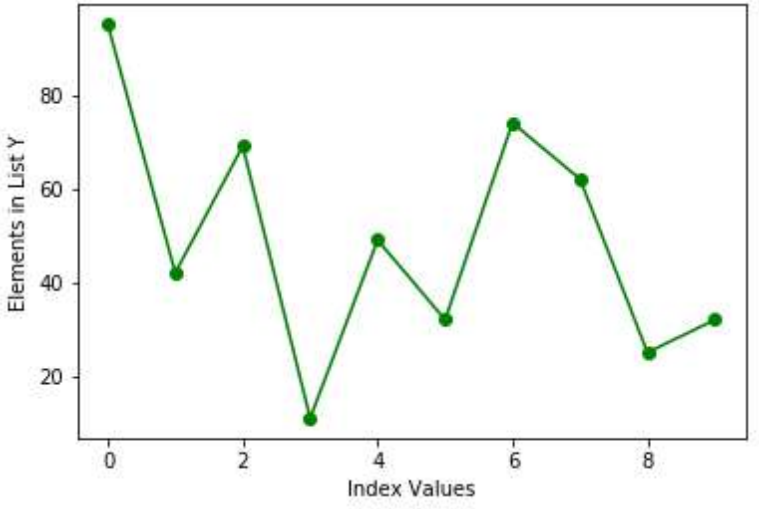

Figure 9: Line plot with circle markers

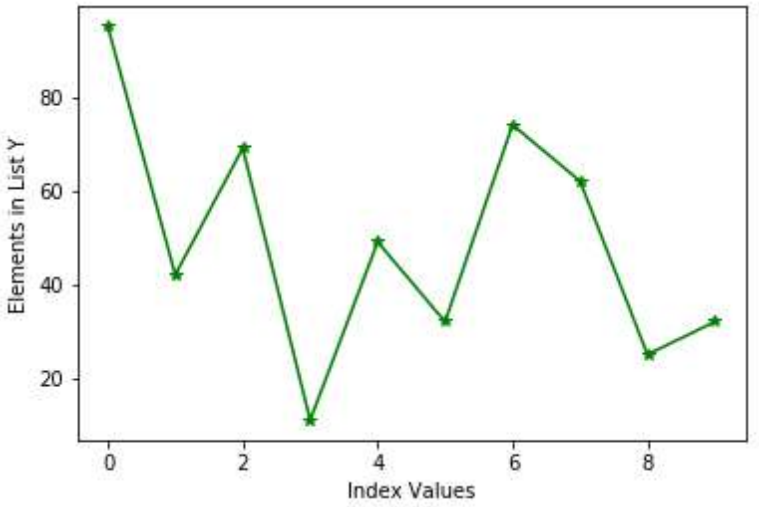

Figure 10: Line chart with asterisk markers

we like. It is also possible to change marker style by tweaking marker parameter in *fmt* string as shown below:

```
# -Example 10-
# Plot continuous green line with asterisk markers
plt.plot(y, 'g*-')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

The output of the above code is *figure 10* where the line and markers share the same color, i.e. green specified by the *fmt* string. If we are to plot line and markers with different colors, we can use multiple `plot` methods to achieve the same.

```
# -Example 11-
# Plot list 'y'
plt.plot(y, 'g')
```

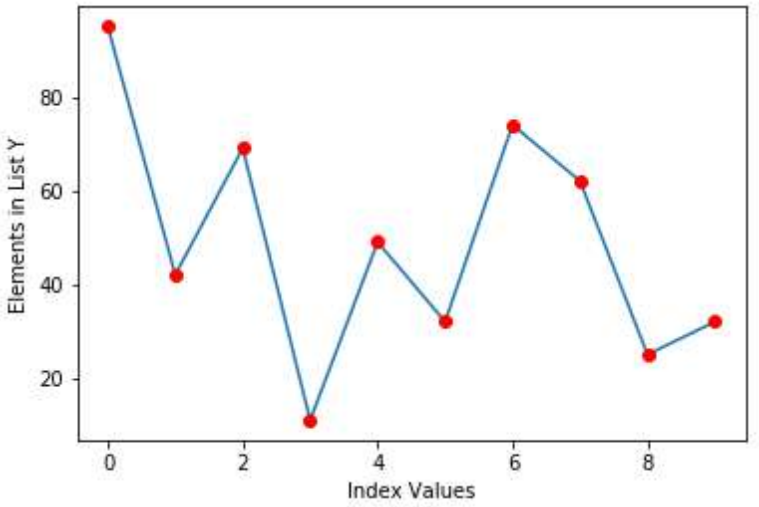

Figure 11: Plot with blue line and red markers

```
# Plot red circle markers
plt.plot(y, 'ro')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

The above code plots line along with *red circle markers* as seen in *figure 11*. Here, we first plot the line with the default style and then attempt to plot *markers* with attributes `r` referring to red color and `o` referring to circle. On the same lines, we can plot multiple sets of data using the same technique. The example given below plots two lists on the same plot.

```
# -Example 12: Technique 1-
# Define two lists
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
y2 = [35, 52, 96, 77, 36, 66, 50, 12, 35, 63]

# Plot lists and show them
plt.plot(y, 'go-')
```

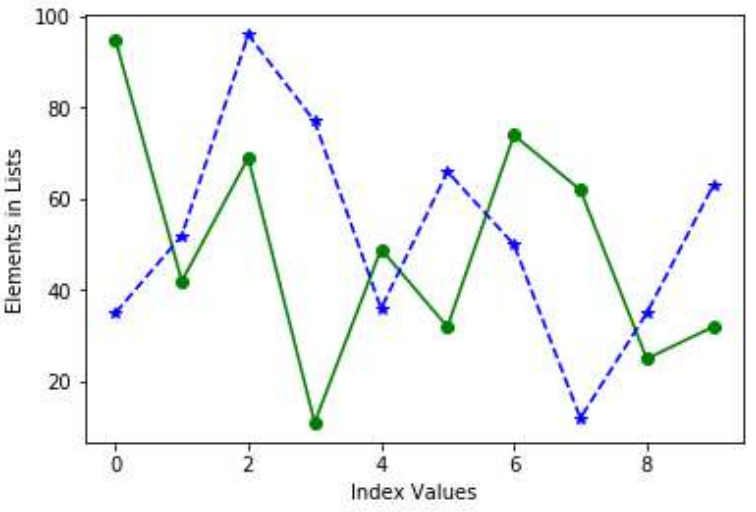

Figure 12: Line plot with two lines

```
plt.plot(y2, 'b*--')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in Lists')
plt.show()
```

The output can be seen in *figure 12* where both green and blue lines are drawn on the same plot. We can achieve the same result as shown above using the different technique as shown below:

```
# -Example 12: Technique 2-
# Plot lists and show them
plt.plot(y, 'go-', y2, 'b*--')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in Lists')
plt.show()
```

Essentially, the `plot` method makes it very easy to plot sequential data

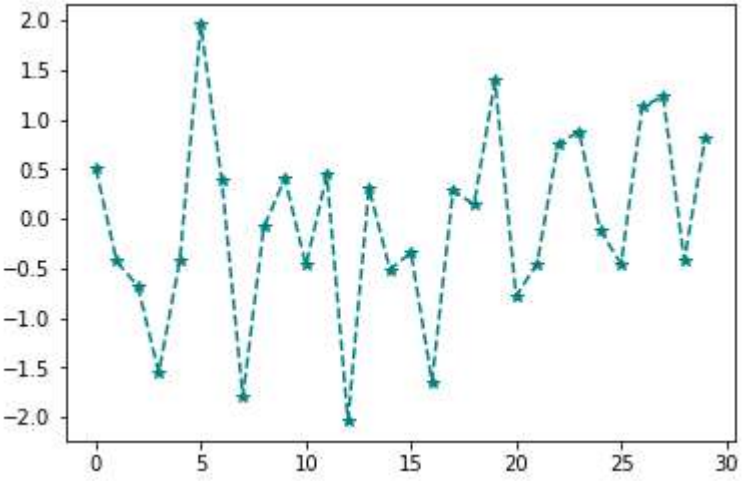

Figure 13: Line plot from a NumPy array

structure such as list, NumPy arrays, pandas series, etc. Similar to plotting lists, we can plot NumPy arrays directly via the `plot` method. Let us plot NumPy one dimensional array. As we are executing codes directly in IPython console, calling the `plt.show()` is not required and hence, we will not be calling the same in subsequent examples. However, remember, it is absolutely necessary to call it while writing Python code in order to show a plot.

```
# -Example 13-
# Importing NumPy library
import numpy as np

# Drawing 30 samples from a standard normal distribution
# into an array 'arr'
arr = np.random.normal(size=30)

# Plotting 'arr' with dashed line-style and * markers
plt.plot(arr, color='teal', marker='*', linestyle='dashed')
```

In the above example, we draw thirty samples from a normal distribution into an array `arr` which in turn gets plotted as a *dashed* line along with

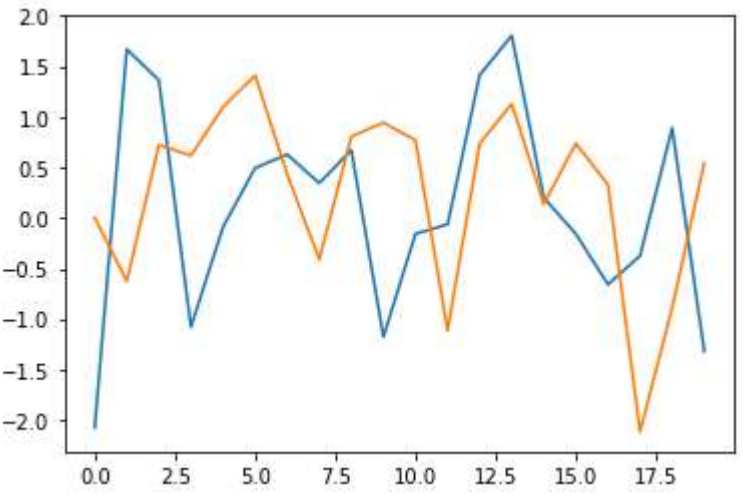

Figure_14: Line plot from 2-D NumPy array

asterisk markers as seen in the *figure 13*.

Plotting two-dimensional arrays follows the same pattern. We provide a 2-D array to a `plot` method to plot it. The below code shows the example of this whose output is shown in *figure 14*.

```
# -Example 14-
# Creating a two dimensional array 'arr_2d' with 40 samples
# and shape of (20, 2)
arr_2d = np.random.normal(size=40).reshape(20, 2)

# Plotting the array
plt.plot(arr_2d)
```

Let us now move our focus to plot pandas data structures. The pandas library uses the standard convention as the matplotlib for plotting directly from its data structures. The pandas also provide a `plot` method which is equivalent to the one provided by matplotlib. Hence, the `plot` method can be called directly from pandas Series and DataFrame objects. The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`. The below example illustrates plotting pandas Series object:

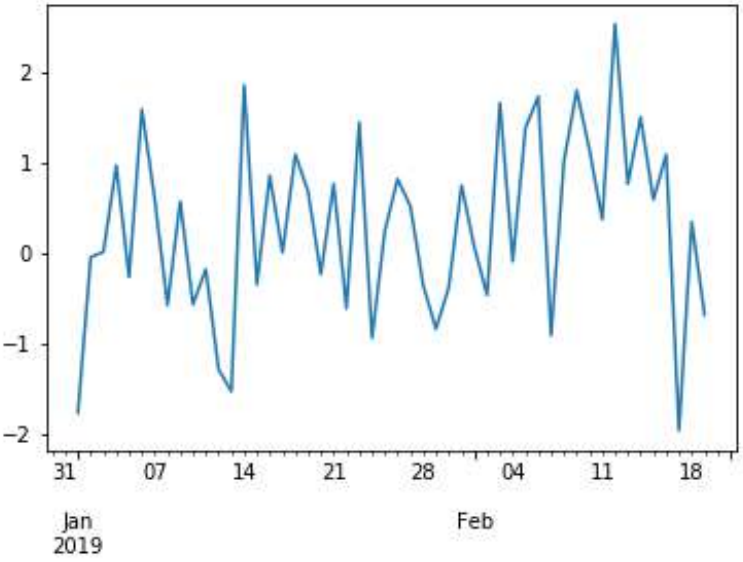

Figure 15: Line plot from a pandas series

```
# -Example 15-
# Importing necessary libraries
import pandas as pd
import numpy as np

# Creating pandas Series with 50 samples drawn from normal
# distribution
ts = pd.Series(np.random.normal(size=50),
               index=pd.date_range(start='1/1/2019',
               periods=50))

# Plotting pandas Series
ts.plot()
```

In the above example, we call the `plot` method directly on pandas Series object `ts` which outputs the plot as shown in *figure 15*. Alternatively, we could have called `plt.plot(ts)`. Calling `ts.plot()` is equivalent to calling `plt.plot(ts)` and both calls would result in almost the same output as shown above. Additionally, the `plot()` method on pandas object sup-

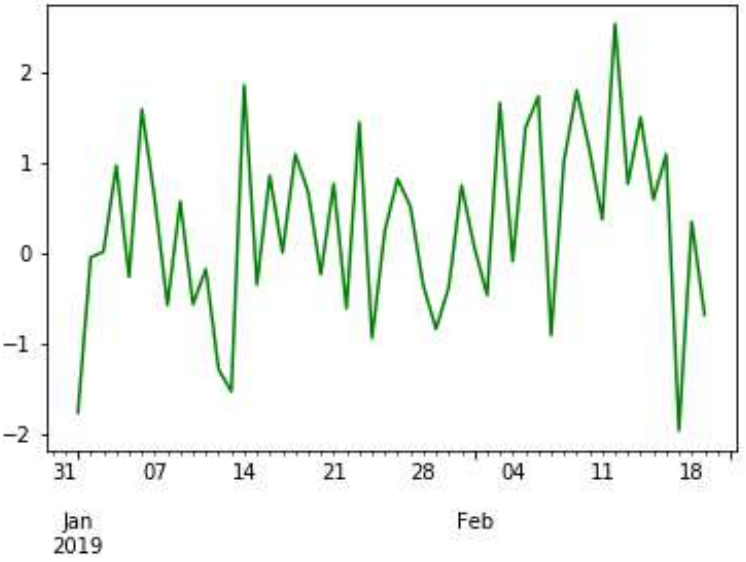

Figure 16: Line plot from a pandas series in green color

ports almost every attribute that `plt.plot()` supports for formatting. For example, calling the `plot` method on pandas objects with a `color` attribute would result in a plot with color mentioned by its value. This is shown below:

```
# -Example 16-
# Plotting pandas Series in green color
ts.plot(color='green')
```

The output of the above code is shown in *figure 16*.

Moving forward, the same notation is followed by pandas DataFrame object and visualizing data within a dataframe becomes more intuitive and less quirky. Before we attempt to plot data directly from a dataframe, let us create a new dataframe and populate it. We fetch the stock data of AAPL ticker that we will be using for illustration purposes throughout the remaining chapter.

```
# -Script to fetch AAPL data from a web resource-
# Import libraries
```

```
import pandas as pd

# Fetch data
data = pd.read_csv('https://bit.ly/2WcsJE7', index_col=0,
                   parse_dates=True)
```

The dataframe `data` will contain stock data with dates being the index. The excerpt of the downloaded data is shown below:

Date	Open	High	Low	Close	Volume	...
2018-03-27	173.68	175.15	166.92	168.340	38962893.0	...
2018-03-26	168.07	173.10	166.44	172.770	36272617.0	...
2018-03-23	168.39	169.92	164.94	164.940	40248954.0	...

Now we can plot any column of a `data` dataframe by calling `plot` method on it. In the example given below, we plot the recent 100 data points from the `Volume` column of the dataframe:

```
# -Example 17-
# Plot volume column
data.Volume.iloc[:100].plot()
```

The output of the above code is shown in *figure 17*. With a dataframe, `plot` method is a convenience to plot all of the columns with labels. In other words, if we plot multiple columns, it would plot labels of each column as well. In the below example, we plot `AdjOpen` and `AdjClose` columns together and the output for the same is shown in *figure 18*.

```
# -Example 18-
data[['AdjOpen', 'AdjClose']][:50].plot()
```

The `plot` method generates a line plot by default when called on pandas data structures. However, it can also produce a variety of other charts as we will see later in this chapter. Having said that, lets head forward to plot scatter plots.

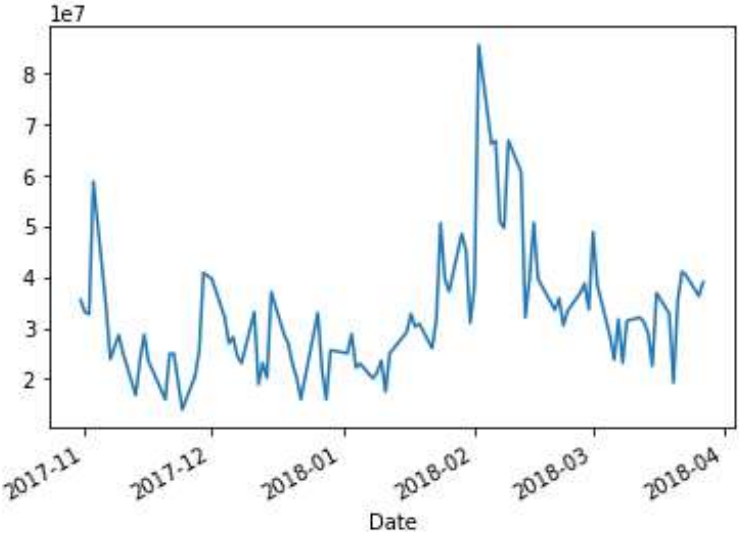

Figure 17: Line plot of a volume column

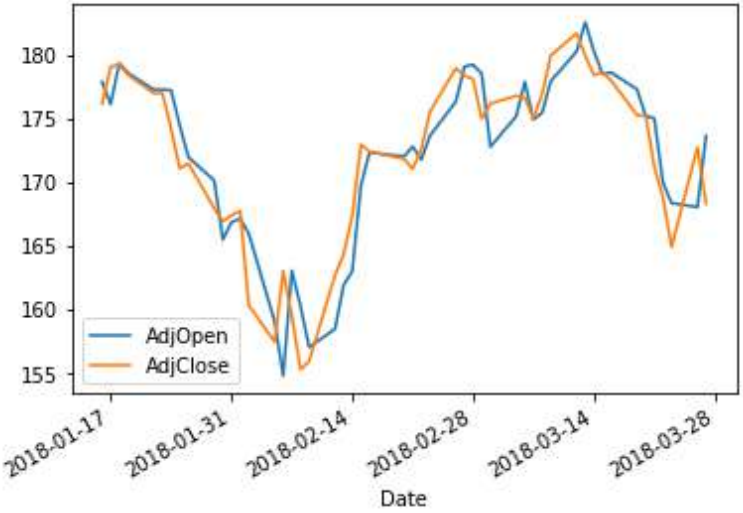

Figure 18: Line plot of two columns

12.2.2 Scatter Plot

Scatter plots are used to visualize the relationship between two different data sets. Matplotlib provides the `scatter` method within `pyplot` sub-module using which scatter plots can be generated.

- `plt.scatter` generates scatter plot of y vs x with varying marker size and/or color.

The x and y parameters are data positions and it can be array-like sequential data structures. There are some instances where we have data in the format that lets us access particular variables with string. For example, Python dictionary or pandas dataframe. Matplotlib allows us to provide such an object with the `data` keyword argument to the `scatter` method to directly plot from it. The following example illustrates this using a dictionary.

```
# -Example 19-
# Creating a dictionary with three key-value pairs
dictionary = {'a': np.linspace(1, 100, 50),
              'c': np.random.randint(0, 50, 50),
              'd': np.abs(np.random.randn(50)) * 100}

# Creating a new dictionary key-value pair
dictionary['b'] = dictionary['a'] * np.random.rand(50)

# Plotting a scatter plot using argument 'data'
plt.scatter('a', 'b', c='c', s='d', data=dictionary)

# Labeling the plot and showing it
plt.xlabel('A Data Points')
plt.ylabel('B Data Points')
plt.show()
```

In the above code, we created a dictionary `dictionary` with four key-value pairs. Values in key `a` and `b` contain fifty random values to be plotted on a scatter plot. Key `c` contains fifty random integers and key `d` contains fifty positive floats which represents color and size respectively for each scatter data point. Then, a call to `plt.scatter` is made along with all keys and the `dictionary` as the value to `data`. The argument `c` within the call refers

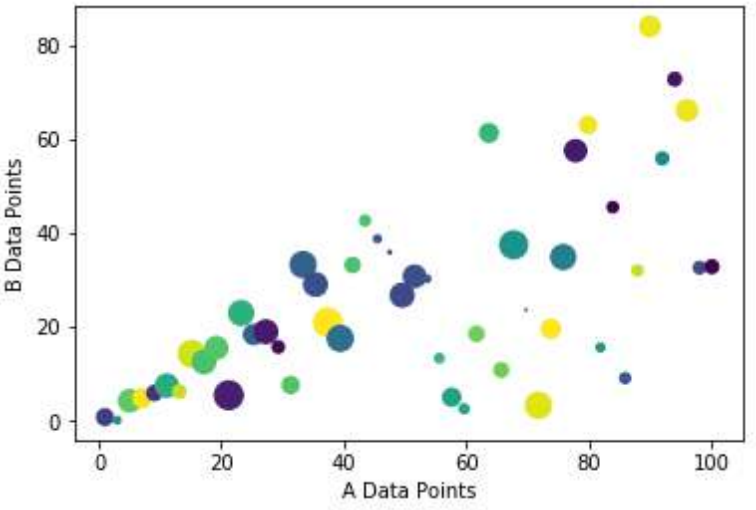

Figure 19: Scatter plot with different size and color

to color to be used and the argument `s` represents the size of a data point. These arguments `c` and `s` are optional. The output we get is a scatter plot with different size and color as shown in *figure 19*. A simple scatter plot with the same color and size gets plotted when we omit these optional arguments as shown in the following example:

```
# -Example 20-
# Creating a scatter plot without color and the same size
plt.scatter(dictionary['a'], dictionary['b'])

# Labeling the plot and showing it
plt.xlabel('A Data Points')
plt.ylabel('B Data Points')
plt.show()
```

The output of the above code will be a scatter plot as shown in *figure 20*. To better understand the working of scatter plots, let us resort to our old friends: lists `x` and `y`. We defined them earlier whem we learned line plots and scatter plots. To refresh our memory, we re-define the same lists below:

```
# Data points for scatter plot
```

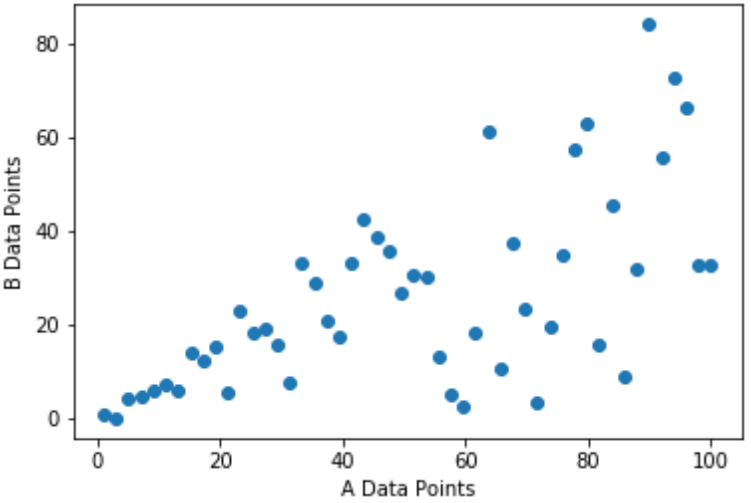

Figure 20: Scatter plot with the same size and color

```
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
```

In addition to these lists, we would be defining two more NumPy arrays `color` and `size` which determines the color and size respectively of each data point while plotting the scatter plot.

```
# Arrays which defines color and size of each data point
color = np.random.rand(10)
size = np.random.randint(50, 100, 10)
```

Now that we have data points ready, we can plot a scatter plot out of them as below:

```
# -Example 21-
# Creating a scatter plot
plt.scatter(x, y, c=color, s=size)

# Labeling the plot and showing it
plt.xlabel('Values from list x')
plt.ylabel('Values from list y')
plt.show()
```

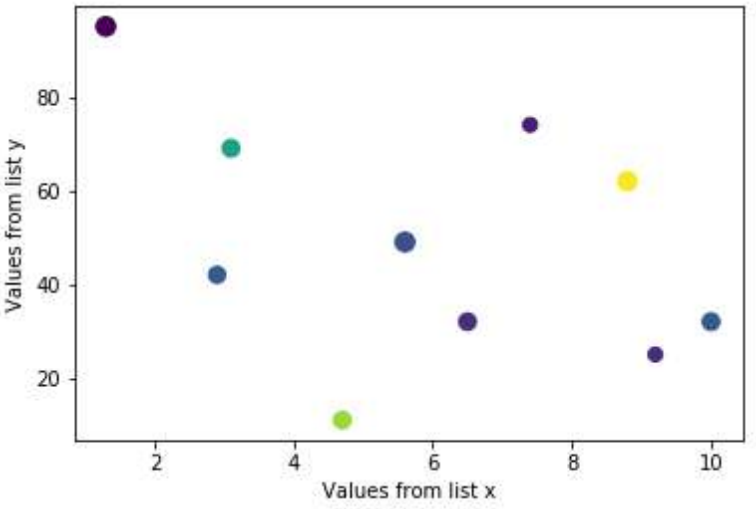

Figure 21: Scatter plot of lists x and y

The scatter plot would contain data points each with different color and size (as they are randomly generated). The output is shown in figure *figure 21*.

In finance, scatter plots are widely used to determine the relations between two data sets visually. With our working knowledge of scatter plots, let's plot `AdjOpen` and `AdjClose` prices of AAPL stock that we have in pandas dataframe `data`. When it comes to plotting data directly from a pandas dataframe, we can almost always resort to `plot` method on pandas to plot all sorts of plots. That is, we can directly use the `plot` method on the dataframe to plot scatter plots akin to line plots. However, we need to specify that we are interested in plotting a scatter plot using the argument `kind='scatter'` as shown below:

```
# -Example 22-
# Plotting a scatter plot of 'AdjOpen' and 'AdjClose' of
# AAPL stock
data.plot(x='AdjOpen', y='AdjClose', kind='scatter')
plt.show()
```

Interestingly, we only need to specify column names of a dataframe `data`

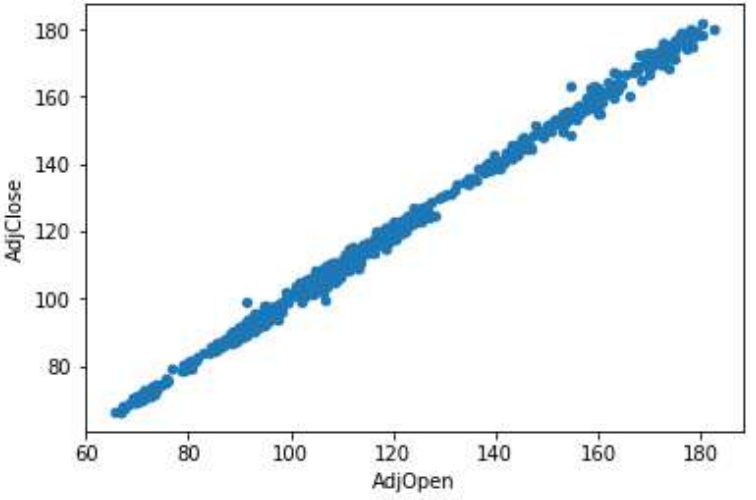

Figure 22: Scatter plot of columns `AdjOpen` and `AdjClose`

for `x` and `y` coordinates along with the argument `kind` which gets resulted in the output as shown in *figure 22*.

By visualizing price patterns using a scatter plot, it can be inferred that open and close prices are positively correlated. Furthermore, we can generate the same plot using the `plt.scatter` method.

```
# Method 1
plt.scatter(x='AdjOpen', y='AdjClose', data=data)
plt.show()

# Method 2
plt.scatter(x=data['AdjOpen'], y=data['AdjClose'])
plt.show()
```

The first method uses the argument `data` which specifies the data source, whereas the second method directly uses dataframe slicing and hence, there is no need to specify the `data` argument.

12.2.3 Histogram Plots

A histogram is a graphical representation of the distribution of data. It is a kind of bar graph and a great tool to visualize the frequency distribution of data that is easily understood by almost any audience. To construct a histogram, the first step is to *bin* the range of data values, divide the entire range into a series of intervals and finally count how many values fall into each interval. Here, the bins are consecutive and non-overlapping. In other words, histograms shows the data in the form of some groups. All the bins/groups go on X-axis, and Y-axis shows the frequency of each bin/group.

The matplotlib library offers a very convenient way to plot histograms. To create a histogram, we use the `hist` method of `pyplot` sub-module of the `matplotlib` library as shown in the below example:

```
# -Example 23-
# Data values for creating a histogram
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Creating a histogram
plt.hist(y)
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

This is the simplest code possible to plot a histogram with minimal arguments. We create a range of values and simply provide it to the `hist` method and let it perform the rest of the things (creating bins, segregating each value to corresponding bin, plotting, etc.). It produces the plot as shown in *figure 23*. The `hist` method also take `bins` as an optional argument. If this argument is specified, bins will be created as per the specified value, otherwise, it will create bins on its own. To illustrate this, we explicitly specify the number of bins in the above code and generate the plot. The modified code and output is shown below:

```
# -Example 24-
# Data values for creating a histogram
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
```

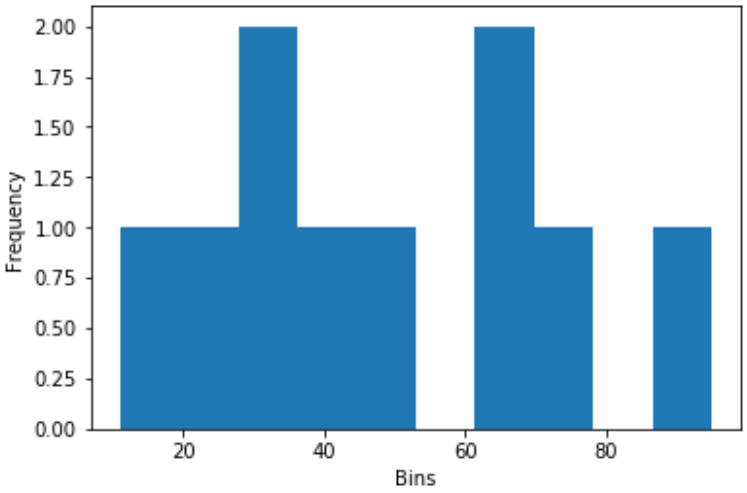

Figure 23: A histogram

```
# Creating a histogram
plt.hist(y, bins= 20)
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

The output we got in *figure 24* is very straight forward. Number `32` appears twice in the list `y` and so it's twice as tall as the other bars on the plot. We specify the number of bins to be `20` and hence, the `hist` method tries to divide the whole range of values into 20 bins and plots them on the X-axis. Similar to the `plot` method, the `hist` method also takes any sequential data structure as its input and plots histogram of it. Let us try to generate a histogram of an array which draws samples from the standard normal distribution.

```
# -Example 25-
# Creating an array
array = np.random.normal(0, 1, 10000)

# Creating a histogram
```

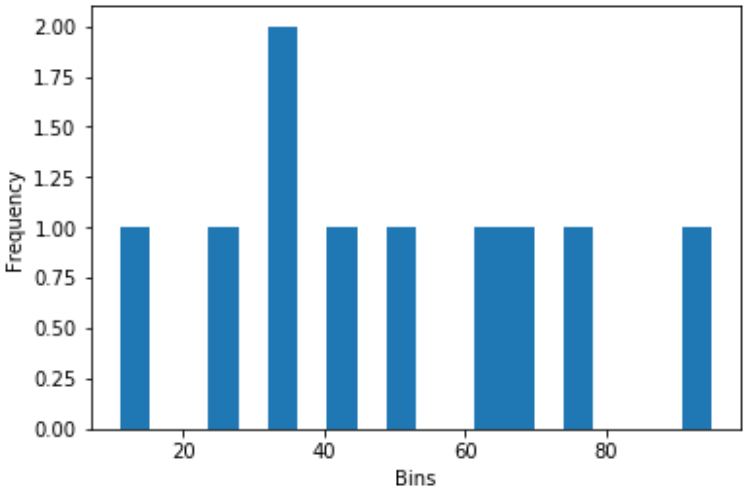

Figure 24: Histogram with 20 bins

```
plt.hist(array)
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

The output we got in *figure 25* shows that the data distribution indeed resembles a normal distribution. Apart from `bins` argument, other arguments that can be provided to `hist` are `color` and `histtype`. There are a number of arguments that can be provided, but we will keep our discussion limited to these few arguments only. The color of a histogram can be changed using the `color` argument. The `histtype` argument takes some of the pre-defined values such as `bar`, `barstacked`, `step` and `stepfilled`. The below example illustrates the usage of these arguments and the output is shown in *figure 26*.

```
# -Example 26-
# Creating an array
array = np.random.normal(0, 1, 10000)

# Creating a histogram and plotting it
plt.hist(array, color='purple', histtype='step')
```

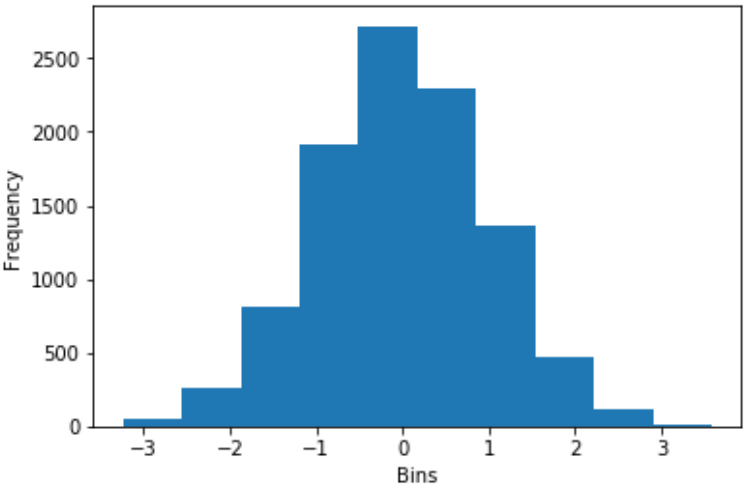

Figure 25: Histogram of an array

```
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

In addition to optional arguments discussed so far, one argument that needs attention is `orientation`. This argument takes either of two values: `horizontal` or `vertical`. The default is `vertical`. The below given example demonstrate the usage of orientation and the output is shown in *figure 27*.

```
# -Example 27-
# Creating an array
array = np.random.normal(0, 1, 10000)

# Creating a histogram and plotting it
plt.hist(array, color='teal', orientation='horizontal')
plt.xlabel('Frequency')
plt.ylabel('Bins')
plt.show()
```

We now shift our focus on plotting a histogram directly from a pandas dataframe. Again, the `plot` method within pandas provides a wrapper

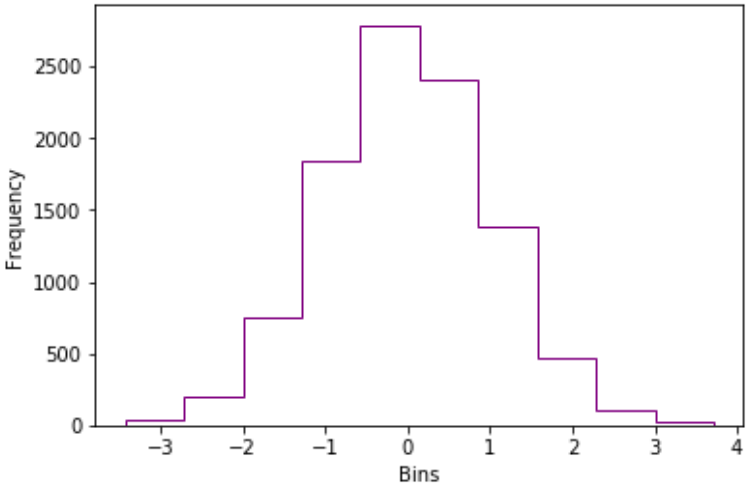

Figure 26: Histogram with `histtype='step'`

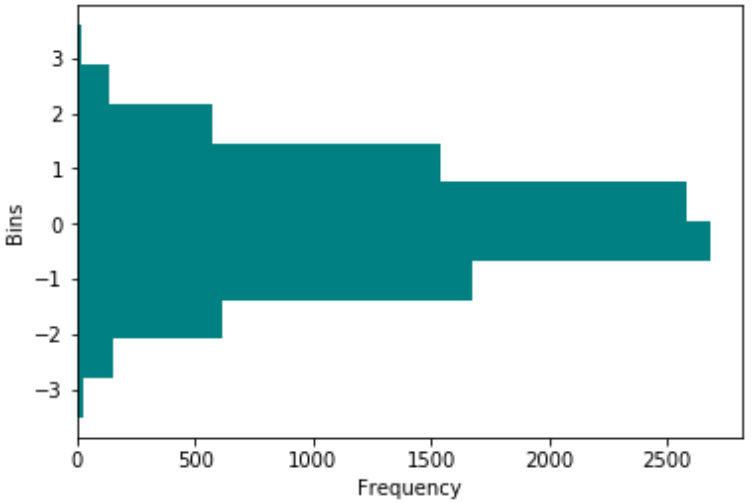

Figure 27: Histogram with horizontal orientation

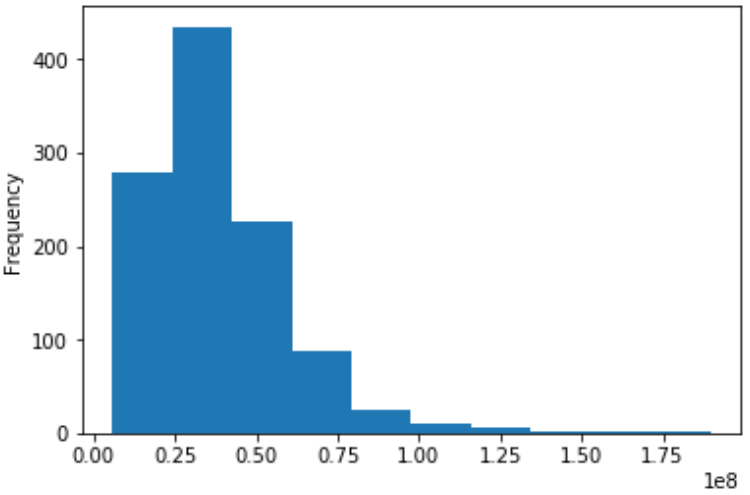

Figure 28: Histogram of a volume column

around the `hist` function in matplotlib as was the case with scatter plots. To plot a histogram, we need to specify the argument `kind` with the value `hist` when a call to `plot` is made directly from the dataframe. We will be working with the same dataframe `data` that contains historical data for AAPL stock.

```
# -Example 28: Technique 1-
# Creating a histogram using a dataframe method
data['Volume'].plot(kind='hist')
plt.show()

# -Example 28: Technique 2-
plt.hist(data['Volume'])
plt.ylabel('Frequency')
plt.show()
```

In the first method, we directly make a call to `plot` method on the dataframe `data` sliced with `Volume` column. Whereas in the second method, we use the `hist` method provided by `matplotlib.pyplot` module to plot the histogram. Both methods plot the same result as shown in *figure 28*.

Figure 29: Line plot of close prices

12.3 Customization

Now that we have got a good understanding of plotting various types of charts and their basic formatting techniques, we can delve deeper and look at some more formatting techniques. We already learned that matplotlib does not add any styling components on its own. It will plot a simple plain chart by default. We, as users, need to specify whatever customization we need. We start with a simple line plot and will keep on making it better. The following example shows plotting of close prices of the AAPL ticker that is available with us in the dataframe `data`.

```
# -Example 29-
# Extracting close prices from the dataframe
close_prices = data['AdjClose']

# Plotting the close prices
plt.plot(close_prices)
plt.show()
```

Here, as shown in *figure 29* the `close_prices` is the pandas Series object which gets plotted using the `plot` method. However, values on the X-axis

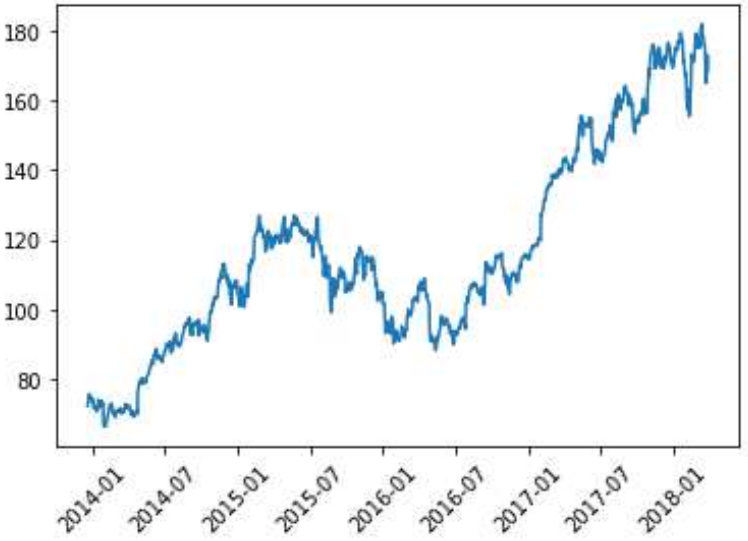

Figure 30: Line plot with rotated `xticks`

are something that we don't want. They are all overlapped with each other. This happens as the `plot` method did not find sufficient space for each date. One way to overcome this issue is to rotate the values on the X-axis to make it look better.

```
# -Example 30-
plt.plot(close_prices)

# Rotating the values along x-axis to 45 degrees
plt.xticks(rotation=45)
plt.show()
```

The `xticks` method along with the `rotation` argument is used to rotate the values/tick names along the x-axis. The output of this approach is shown in *figure 30*. Another approach that can be used to resolve the overlapping issue is to increase the figure size of the plot such that the matplotlib can easily show values without overlapping. This is shown in the below example and the output is shown in *figure 31*:

```
# -Example 31-
# Creating a figure with the size 10 inches by 5 inches
```

Figure 31: Line plot with custom figure size

```
fig = plt.figure(figsize=(10, 5))
plt.plot(close_prices)
plt.show()
```

Similarly, the matplotlib provides `yticks` method that can be used to customize the values on the Y-axis. Apart from the `rotation` argument, there are a bunch of other parameters that can be provided `xticks` and `yticks` to customize them further. We change the font size, color and orientation of ticks along the axes using the appropriate arguments within these methods in the following example:

```
# -Example 32-
# Creating a figure, setting its size and plotting close
# prices on it
fig = plt.figure(figsize=(10, 5))
plt.plot(close_prices, color='purple')

# Customizing the axes
plt.xticks(rotation=45, color='teal', size=12)
plt.yticks(rotation=45, color='teal', size=12)

# Setting axes labels
plt.xlabel('Dates', {'color': 'orange', 'fontsize':15})
plt.ylabel('Prices', {'color': 'orange', 'fontsize':15})
plt.show()
```

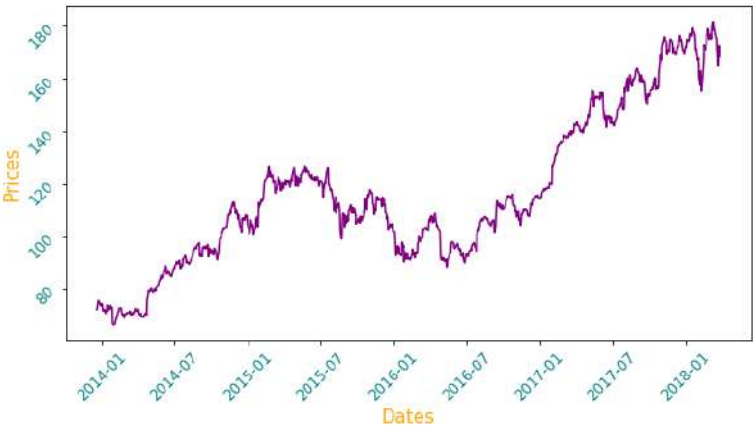

Figure 32: Line plot with rotated ticks on axes and colored values

Along with the axes values, we change the color and font size of axes labels as shown in *figure 32*. There are numbers of other customizations possible using various arguments and matplotlib provides total flexibility to create the charts as per one's desire. Two main components that are missing in the above plot are title and legend, which can be provided using the methods `title` and `legends` respectively. Again, as with the other methods, it is possible to customize them in a variety of way, but we will be restricting our discussion to a few key arguments only. Adding these two methods as shown below in the above code would produce the plot as shown in *figure 33*:

```
# -Example 33-
# Showing legends and setting the title of plot
plt.legend()
plt.title('AAPL Close Prices', color='purple', size=20)
```

Another important feature that can be added to a figure is to draw a grid within a plot using the `grid` method which takes either `True` or `False`. If true, a grid is plotted, otherwise not. An example of a plot with grid is shown below and its output is shown in *figure 34*.

```
# -Example 34-
# Adding the grid to the plot
plt.grid(True)
```

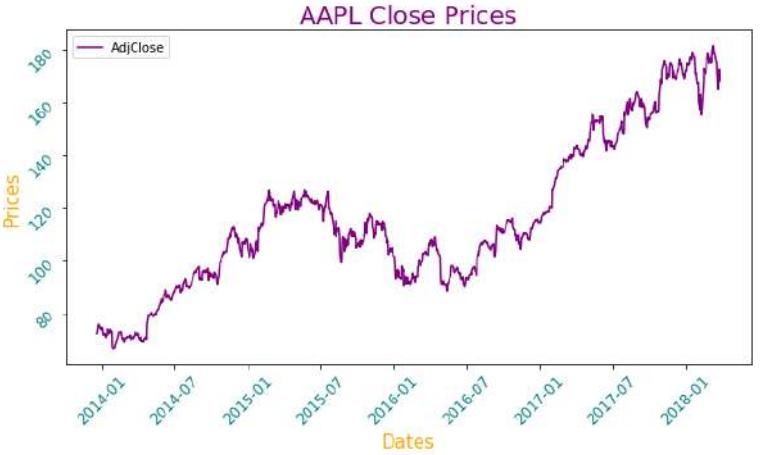

Figure 33: Line plot with legends and the title

The `axhline` method allows us to add a horizontal line across the axis to the plot. For example, we might consider adding the mean value of close prices to show the average price of a stock for the whole duration. It can be added using `axhline` method. Computation of mean value and its addition to the original plot is shown below:

```
# -Example 35-
# Importing NumPy library
import numpy as np

# Calculating the mean value of close prices
mean_price = np.mean(close_prices)

# Plotting the horizontal line along with the close prices
plt.axhline(mean_price, color='r', linestyle='dashed')
```

Now that we have the mean value of close prices plotted in the *figure 35*, one who looks at the chart for the first time might think what this red line conveys? Hence, there is a need to explicitly mention it. To do so, we can use the `text` method provided by `matplotlib.pyplot` module to plot text anywhere on the figure.

```
# -Example 36-
```

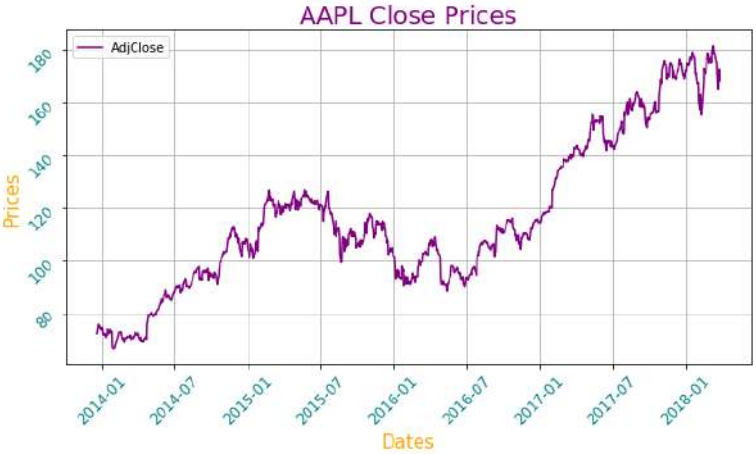

Figure 34: Line plot with a grid

Figure 35: Line plot with horizontal line

Figure 36: Line plot with text on it

```
# Importing DateTime from DateTime library
from datetime import datetime

# Plotting text on date 2014-1-7 and price 120
plt.text(datetime(2014,1,7), 120, 'Mean Price',
size=15, color='r')
```

The `text` method takes three compulsory arguments: `x`, `y` and `t` which specifies the coordinates on X and Y-axis and text respectively. Also, we use a `datetime` sub-module from a `datetime` library to specify a date on the X-axis as the plot we are generating has dates on the X-axis. The chart with text indicating the mean price is shown in *figure 36*.

Using all these customization techniques, we have been able to evolve the dull looking price series chart to a nice and attractive graphic which is not only easy to understand but presentable too. However, we have restricted ourselves to plotting only a single chart. Let us brace ourselves and learn to apply these newly acquired customization techniques to multiple plots.

We already learned at the beginning of this chapter that a figure can have multiple plots, and that can be achieved using the `subplots` method. The following examples show stock prices of AAPL stock along with its traded

volume on each day. We start with a simple plot that plots stock prices and volumes in the below example:

```
# -Example 37-
# Extracting volume from the dataframe 'data'
volume = data['AdjVolume']

# Creating figure with two rows and one column
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
                               sharex=True,
                               figsize=(10, 8))

# Plotting close prices on the first sub-plot
ax1.plot(close_prices, color='purple')
ax1.grid(True)

# Plotting trading volume on the second sub-plot
ax2.bar(volume.index, volume)
ax2.grid(True)

# Displaying the plot
plt.show()
```

First, we extract the `AdjVolume` column from the `data` dataframe into a `volume` which happens to be pandas series object. Then, we create a figure with sub-plots having two rows and a single column. This is achieved using `nrows` and `ncols` arguments respectively. The `sharex` argument specifies that both sub-plots will share the same x-axis. Likewise, we also specify the figure size using the `figsize` argument. These two subplots are unpacked into two axes: `ax1` and `ax2` respectively. Once, we have the axes, desired charts can be plotted on them.

Next, we plot the `close_prices` using the `plot` method and specify its color to be `purple` using the `color` argument. Similar to the `plot` method, matplotlib provides `bar` method to draw bar plots which takes two arguments: the first argument to be plotted on the X-axis and second argument to be plotted along the y-axis. For our example, values on X-axis happens to be a date (specified by `volume.index`), and value for each bar on the Y-axis is provided using the recently created `volume` series. After that, we plot grids

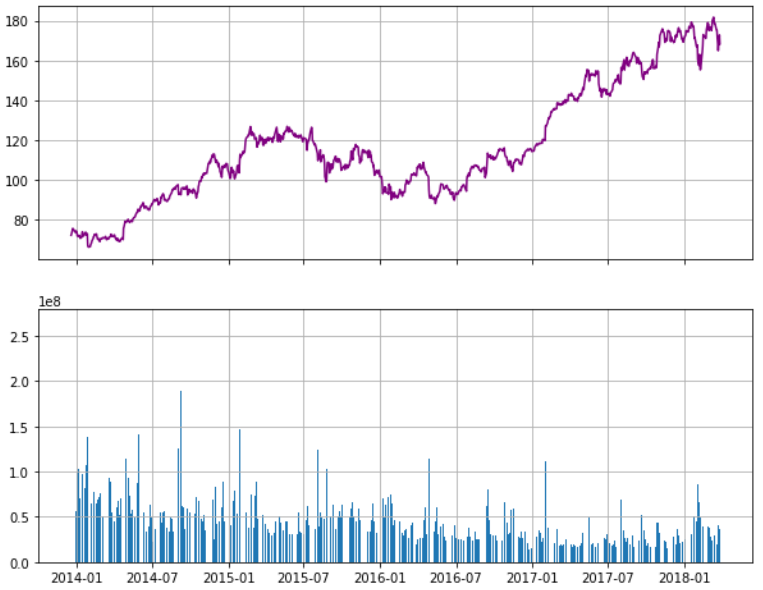

Figure 37: Sub-plots with stock price and volume

on both plots. Finally, we display both plots. As can be seen above in *figure 37*, matplotlib rendered a decent chart. However, it misses some key components such as title, legends, etc. These components are added in the following example:

```
# -Example 38-
# Creating figure with multiple plots
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
                               sharex=True,
                               figsize=(10, 8))
ax1.plot(close_prices, color='purple', label='Prices')
ax1.grid(True)

# Setting the title of a first plot
ax1.set_title('Daily Prices')

# Setting the legend for the first plot
ax1.legend()
```

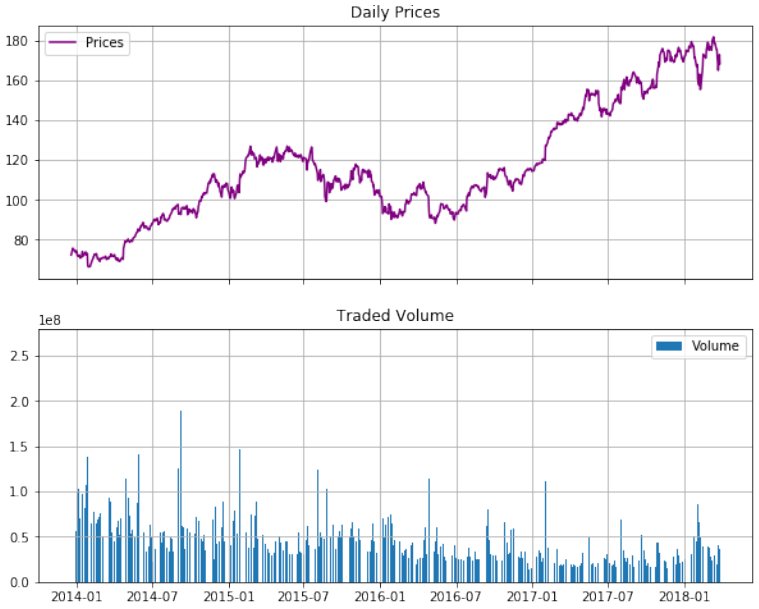

Figure 38: Sub-plots with legends and titles

```
ax2.bar(volume.index, volume, label='Volume')
ax2.grid(True)

# Setting the title of a second plot
ax2.set_title('Volume')

# Setting the legend for the second plot
ax2.legend()

plt.show()
```

Here, we use the `legend` method to set legends in both plots as shown in *figure 38*. Legends will print the values specified by the `label` argument while plotting each plot. The `set_title` is used to set the title for each plot. Earlier, while dealing with the single plot, we had used the `title` method to set the title. However, it doesn't work the same way with multiple plots.

Another handy method provided by the matplotlib is the `tight_layout`

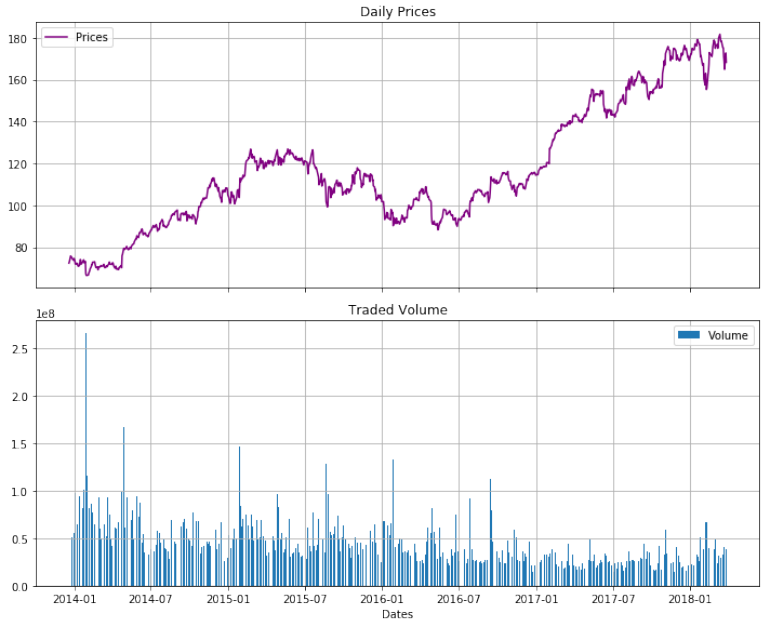

Figure 39: Sub-plots with tight layout

method which automatically adjusts the padding and other similar parameters between subplots so that they fits into the figure area.

```
# -Example 39-
# Setting layout
plt.tight_layout()

# Setting label on the x-axis
plt.xlabel('Dates')
plt.show()
```

The above code explicitly specifies the layout and the label on the x-axis which results into the chart as shown in *figure 39*.

In addition to all this customization, matplotlib also provides a number of predefined styles that can be readily used. For example, there is a predefined style called "ggplot", which emulates the aesthetics of *ggplot* (a popular plotting package for R language). To change the style of plots being

rendered, the new style needs to be explicitly specified using the following code:

```
plt.style.use('ggplot')
```

One the style is set to use, all plots rendered after that will use the same and newly set style. To list all available styles, execute the following code:

```
plt.style.available
```

Let us set the style to one of the pre-defined styles known as *'fivethirtyeight'* and plot the chart.

```
# -Example 40-
plt.style.use('fivethirtyeight')
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
                               sharex=True,
                               figsize=(10, 8))

ax1.plot(close_prices, color='purple', label='Prices')
ax1.grid(True)
ax1.set_title('Daily Prices')
ax1.legend()

ax2.bar(volume.index, volume, label='Volume')
ax2.grid(True)
ax2.set_title('Traded Volume')
ax2.legend()

plt.tight_layout()

plt.xlabel('Dates')
plt.show()
```

The output of the above code is shown in the *figure 40*. By changing the style, we get a fair idea about how styles play an important role to change the look of charts cosmetically while plotting them.

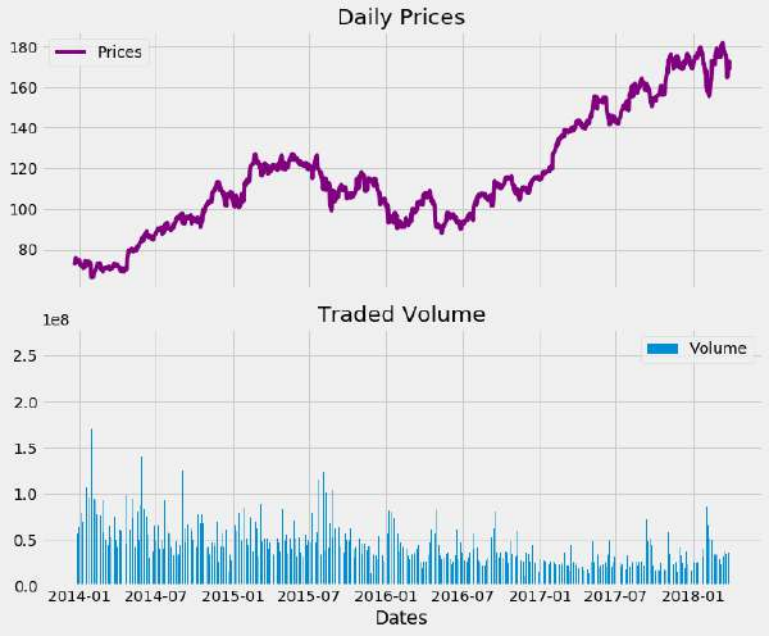

Figure 40: Plot with pre-defined style `'fivethirtyeight'`

The last method that we will study is the `savefig` method that is used to save the figure on a local machine. It takes the name of the figure by which it will be saved. This is illustrated below:

```
plt.savefig('AAPL_chart.png')
```

Executing the above code will save the chart we plotted above with the name `AAPL_chart.png`.
This brings us to the end of this chapter. We started with the basics of figure and plots, gradually learning various types of charts and along with the finer details.

We also learned customization and took a sneak peek into plotting multiple plots within the same chart.

12.4 Key Takeaways

1. Matplotlib does not fall under the Python Standard Library. Hence, it needs to be installed before it can be used.
2. The pyplot module within matplotlib provides the common charting functionality and is used to plot various kinds of charts. A common practice is to import it using the alias `plt`.
3. The code `%matplotlib inline` is used to enable plotting charts within Jupyter Notebook.
4. Figure is the top-level container and Axes is the area where plotting occurs. `plt.figure()` creates an empty figure with axes. The `figsize` argument is used to specify the figure size.
5. The `show()` method of pyplot sub-module is used to display the plot.
6. Methods available on the axes object can also be called directly using the `plt` notation.
7. The `plot()` method from pyplot module is used to plot line chart from a sequential data structures such as lists, tuples, NumPy arrays, pandas series, etc.
8. The `scatter()` method from pyplot module is used to generate scatter plots.
9. The `hist()` method from pyplot module is used to generate histogram plots.
10. The `bar()` method from pyplot module is used to create a bar chart.

11. `xlabel()` and `ylabel()` methods are used to label the charts. The `title()` method is used to title the chart.
12. The `legend()` method is used to annotate the charts.
13. The `axhline()` is used to add a horizontal line across the axis to a plot.
14. `xticks()` and `yticks()` methods are used to customize the ticks along the axes in a plot.
15. The `grid()` method is used to add a grid to a plot.
16. The `subplots()` method is used to create multiple plots within a figure.
17. The `style` attribute is used to configure the style of a plot.
18. Methods discussed in this chapter have their own attributes specific to each method.

References

Chapter 1 - Introduction

1. A Byte of Python, Swaroop CH:
   https://python.swaroopch.com/about_python.html
2. Hilpisch, Yves (2014): *"Python for Finance - ANALYZE BIG FINANCIAL DATA."* O'Reilly.
3. Tutorials Point:
   https://www.tutorialspoint.com/python/python_overview.htm
4. Python Official Documentation:
   https://www.python.org/about/apps/
5. Geeks for Geeks:
   https://www.geeksforgeeks.org/important-differences-between-python-2-x-and-python-3-x-with-examples/

Chapter 2 - Getting Started with Python

1. Zhang, Y (2015): *"An Introduction to Python and Computer Programming"*. Springer, Singapore.
2. Python Docs:
   https://docs.python.org/3/tutorial/introduction.html

Chapter 3 - Variables and Data Types in Python

1. A Byte of Python, Swaroop CH:
   https://python.swaroopch.com/about_python.html
2. W3 Schools:
   https://www.w3schools.com/python/python_ref_string.asp
3. Tutorials Point:
   https://www.tutorialspoint.com/python/python_strings.htm

Chapter 4 - Modules, Packages and Libraries

1. Python Official Documentation:
   https://docs.python.org/3/tutorial/modules.html and
   https://docs.python.org/3/library/index.html

Chapter 5 - Data Structures

1. Python Official Documentation:
   https://docs.python.org/3/tutorial/datastructures.html

Chapter 6 - Keywords & Operators

1. W3 Schools:
   https://www.w3schools.com/python/

Chapter 7 - Control Flow Statements

1. W3 Schools:
   https://www.w3schools.com/python/ref_func_range.asp
2. Programiz:
   https://www.programiz.com/python-programming/list-comprehension

Chapter 8 - Iterators & Generators

1. Iterators, generators and decorators:
   https://pymbook.readthedocs.io/en/latest/igd.html
2. W3 Schools:
   https://www.w3schools.com/python/python_iterators.asp

Chapter 9 - Functions in Python

1. Python Official Documentation:
   https://docs.python.org/3/tutorial/classes.html
2. A Byte of Python, Swaroop CH:
   https://python.swaroopch.com/
3. w3schools.com:
   https://www.w3schools.com/python/python_ref_functions.asp
4. Programiz:
   https://www.programiz.com/python-programming/

Chapter 10 - Numpy Module

1. Towards Data Science:
   https://towardsdatascience.com/a-hitchhiker-guide-to-python-numpy-arrays-9358de570121
2. NumPy documentation:
   https://docs.scipy.org/doc/numpy-1.13.0/

Chapter 11 - Pandas Module

1. 10 Minutes to pandas:
   https://pandas.pydata.org/pandas-docs/stable/10min.html
2. Pandas IO Tools:
   https://pandas.pydata.org/pandas-docs/stable/io.html

Chapter 12 - Data Visualization with Matplotlib

1. Matplotlib Plot:
   https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html
2. Pyplot Tutorial:
   https://matplotlib.org/tutorials/introductory/pyplot.html

QuantInsti® is one of the pioneer algorithmic trading research and training institutes across the globe. With its educational initiatives, QuantInsti is preparing financial market professionals for the contemporary field of algorithmic and quantitative trading. QuantInsti has also designed education modules and conducted knowledge sessions for/with various exchanges in South and South-East Asia and for leading educational and financial institutions.

QuantInsti's flagship programme 'Executive Programme in Algorithmic Trading' (EPAT®) is designed for professionals looking to grow in the field algorithmic and quantitative Trading. It inspires individuals towards a successful career by focusing on derivatives, quantitative trading, electronic market-making, financial computing and risk management. This comprehensive certificate offers unparalleled insights into the world of algorithms, financial technology and changing market microstructure with its exhaustive course curriculum designed by leading industry experts and market practitioners.

Quantra® is an e-learning portal by QuantInsti that specializes in short self-paced courses on algorithmic and quantitative trading. Quantra offers an interactive environment which supports 'learning by doing' through guided coding exercises, videos and presentations in a highly interactive fashion through machine enabled learning.

Quantra Blueshift® is a comprehensive trading and strategy development platform that lets you focus more on the strategy and less on coding and data. Our cloud-based backtesting engine helps you develop, test and analyse trading strategies and fine-tune them, for free. Apart from imparting knowledge on advanced concepts through its various courses, QuantInsti contributes to the industry through various initiatives including participating in & hosting webinars, conferences and workshops in different parts of the world.

QuantInsti Quantitative Learning Pvt. Ltd.

A-309, Boomerang, Chandivali Farm Road, Powai, Mumbai – 400 072
Email: contact@quantinsti.com