Python for Data Science

NumPy

Introduction

In this module, you will learn about the basics of NumPy, which is the fundamental package for scientific computing in Python. NumPy consists of a robust data structure called multidimensional arrays. Matplotlib, which is used along with NumPy, gives you the ability to visualise data using meaningful plots.
In this session, you will understand the advantages of using NumPy (over standard Python ways). You will also learn how to:

- Create NumPy arrays,
- Convert lists and tuples to NumPy arrays,
- Inspect the structure and content of arrays, and
- Subset, slice, index and iterate through arrays.

Before getting into the technicalities of a NumPy array, exploring its useful functions and understanding how it is implemented in Python, it is crucial to understand why NumPy is an important library for working with data.

NumPy, an acronym for the term ‘Numerical Python’, is a library in Python used extensively for efficient mathematical computing. This library allows users to store large amounts of data using lesser memory and perform extensive operations efficiently. It provides optimised and simpler functionalities to perform aforementioned operations using homogeneous one-dimensional and multi-dimensional arrays (You will learn more about this later.). Now, prior to delving deep into the concepts of NumPy arrays, it is important to note that Python lists can very well perform all the actions that are performed by NumPy arrays; it is simply the fact that NumPy arrays are faster and more convenient than lists when it comes to extensive computations, which make them extremely useful, especially when you are working with large amounts of data.

Basics of NumPy

NumPy is a library written for scientific computing and data analysis. It stands for 'Numerical Python'.
The most basic object in NumPy is a **NumPy array**. It includes the following features:

1. High-performance multi-dimensional array
2. Integration with C/C++
3. Linear Algebra functions
4. Fourier series
5. Random number generator

A list has a limitation, i.e., the inability to operate on the entire data together. But, with the help of a NumPy array, we can operate on the entire data together.

Creating NumPy Arrays

There are two ways to create NumPy arrays as mentioned below.

1. **Converting** the existing lists or tuples to arrays using np.array
2. **Initialising** fixed-length arrays using the NumPy functions

The key advantage of NumPy arrays over lists is that they allow you to operate over the entire data, unlike lists. However, in terms of structure, NumPy arrays are extremely similar to lists. If you try to run the print() command over a NumPy array, then you will get the following output:

[element_1 element_2 element_3…]

The only difference between a NumPy array and a list is that the elements in a NumPy array are separated with a space instead of a comma. This is an aesthetic feature that differentiates a list and a NumPy array. (An important point to note here is that the array given above is a one-dimensional array.)

Before using NumPy, you have to import it.
The NumPy library can be imported as shown in the code snippet given below.

```
import numpy as np
```

To create an array, you need to write the code snippet given below.

```
import numpy as np
np_l = np.array([1, 4, 7])
np_l * 2.5
```

```
array([ 2.5, 10. , 17.5])
```

Another feature of NumPy arrays is that they are **homogeneous** in nature. By homogeneous, we mean that all the elements in a NumPy array have to be of the same data type, which could be an integer, a float, a string, etc. In the code snippet below, we can see that a NumPy array converts all the values to one type: string.

```
np.array([1, "avc", True])
```

```
array(['1', 'avc', 'True'], dtype='<U11')
```

Also, when we try to add two lists, it concatenates, but when we add two NumPy arrays, then it adds the respective indexes as shown below.

```
list1 = [1, 4, 6]
list2 = [3, 6, 7]
list1 + list2
```

```
[1, 4, 6, 3, 6, 7]
```

```
np_1 = np.array(list1);
np_2 = np.array(list2);
np_1 + np_2
```

```
array([ 4, 10, 13])
```

Operations Over 1-D Arrays

Once you have loaded the data into an array, NumPy offers a wide range of operations to perform on the data.

Let's take an example. A panel wants to select cricketers for an upcoming league match based on their fitness. Players from all significant cricket clubs participated in a practice match, and their data is collected. Now, let's explore the features of NumPy using the players' data.

The heights of the players are stored as a regular Python list: height_in and expressed in inches. The weights of the players are stored as a regular Python list: weight_lb and expressed in pounds. We first need to store the data into a NumPy array.

```
heights_in = [74, 74, 72, 72, 73, 69, 69, 71, 76, 71, 73, 73, 74, 74, 69, 70, 73, 75, 78, 79, 76, 74, 76, 72, 71, 75, 77, 74, 73,
weights_lb = [180, 215, 210, 210, 188, 176, 209, 200, 231, 180, 188, 180, 185, 160, 180, 185, 189, 185, 219, 230, 205, 230, 195,

import numpy as np
heights = np.array(heights_in)
weights = np.array(weights_lb)
```

To see the type of data stored, we can use the following type function:

```
type(heights)
```

```
numpy.ndarray
```

To see the size or length of an array, we can use the following size method:

```
heights.size
```

```
1015
```

To calculate the BMI of the players, we first need to convert the heights to metres and the weights to kilograms as given below.

```
heights_m = heights *0.025
weights_kg = weights * 0.45
heights_m
```

```
array([1.85 , 1.85 , 1.8  , ..., 1.875, 1.875, 1.825])
```

Then, we need to calculate the BMI as follows:

```
bmi = weights_kg / (heights_m **2)
bmi
```

```
array([23.66691015, 28.26880935, 29.16666667, ..., 26.24      ,
       24.32      , 26.34640646])
```

In the example given above, if we had been working with lists, then we would have to map a lambda function (or worse, write a for loop). Whereas with NumPy, we simply need to multiply the arrays.

For one-dimensional arrays, indexing, slicing, etc. is similar to Python lists, which means that indexing starts at 0.

To access the BMI of the fifth player:

```
bmi[4]
```

```
25.400638018389937
```

To access the BMI of the second last player:

```
bmi[-2]
```

```
24.32
```

To access the BMI of the first five players:

```
bmi[:5]
```

```
array([23.66691015, 28.26880935, 29.16666667, 29.16666667, 25.40063802])
```

To access the BMI of the last three players:

```
bmi[-3:]
```

```
array([26.24      , 24.32      , 26.34640646])
```

To check how many players are underweight (i.e., their BMI is less than 21):

```
#see underweight players
bmi < 21
```

```
array([False, False, False, ..., False, False, False])
```

```
bmi[bmi < 21]
```

```
array([20.83333333, 20.83333333, 20.64429077, 19.968     ])
```

```
bmi[bmi < 21].size
```

```
4
```

To calculate the maximum, minimum and average BMI:

```
bmi.max(), bmi.min(), bmi.mean()
```

```
(36.11111111111111, 19.968, 26.684334976283704)
```

Multidimensional Arrays

A **multidimensional array** is an array of arrays. For example, a two-dimensional array would be an array with each element as a one-dimensional array.

A one-dimensional and multidimensional arrays are printed as follows:

1-D array: [1, 2, 3, 4, 5]

2-D array: [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]

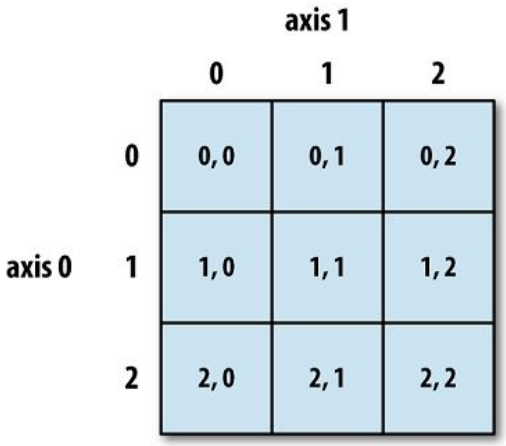
3-D array: [[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
[[10, 11, 12], [13, 14, 15], [16, 17, 18]],
[[19, 20, 21], [22, 23, 24], [25, 26, 27]]]

Similarly, a three-dimensional array can be thought of as an array with each element as a two-dimensional array. To create multidimensional arrays, you can give a multidimensional list as an input to the np.array function.

In NumPy, dimensions are called **axes**. In NumPy terminology, for 2-D arrays:

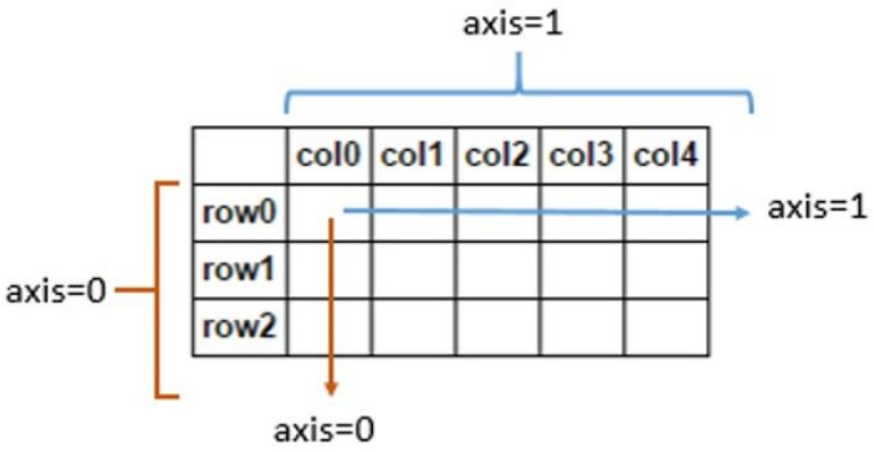
- *axis = 0* refers to the rows, and
- *axis = 1* refers to the columns.

The coordinates of a two-dimensional array are as follows:



Multidimensional Array

Multidimensional arrays are indexed using as many indices as the number of dimensions or axes. For instance, to index a 2-D array, you need two indices: array[x, y]. Each axis has an index starting at 0. The figure provided above shows the axes and their indices for a 2-D array.



Now, let's consider the example discussed above to create a 2-D array.
The player information is provided in the following two lists.
Players: A list of tuples where the first element is height in inches and the second element is weight in lbs
Skills: The skill of the players

```
import numpy as np
players = [(74, 180), (74, 215), (72, 210), (72, 210), (73, 188), (69, 176), (69, 209), (71, 200), (76, 231), (71, 180), (73, 188
skills = np.array(['Keeper', 'Batsman', 'Bowler', 'Keeper-Batsman', 'Batsman', 'Keeper-Batsman', 'Batsman', 'Batsman', 'Batsman',
np_players = np.array(players)
np_players
```

```
array([[ 74, 180],
       [ 74, 215],
       [ 72, 210],
       ...,
       [ 75, 205],
       [ 75, 190],
       [ 73, 195]])
```

```
type(np_players)
```

```
numpy.ndarray
```

```
np_players.shape
```

```
(1015, 2)
```

```
np_players.ndim
```

```
2
```

```
np_players.dtype
```

```
dtype('int32')
```

In the snippet given above, we used ‘shape’ to print the NumPy array’s dimensions, and to check the number of dimensions that the array is split into, we used ‘ndim’. Here, it is split into two dimensions: x-axis and y-axis.

```
np_players[0] # to get inormation about 1st player
```

```
array([ 74, 180])
```

```
np_players[0][1] #get weight of 1st player,
```

```
180
```

```
np_players[:, 0] # all players height
```

```
array([74, 74, 72, ..., 75, 75, 73])
```

```
np_players[:, 0]> 75 # this give array with bollean values true for height>75
np_players[np_players[:, 0]> 75] #gives all the players whose height>75
```

```
array([[ 76, 231],
       [ 78, 219],
       [ 79, 230],
       [ 76, 205],
       [ 76, 195],
       [ 77, 203],
       [ 78, 200],
       [ 76, 200],
       [ 77, 220],
       [ 76, 212],
       [ 76, 224],
       [ 78, 210],
       [ 76, 195],
       [ 77, 200],
       [ 81, 260],
       [ 78, 228],
       [ 77, 200],
       [ 76, 190],
       [ 76, 230],
       [ 78, 235]
```

```
#to get information on players whose skill is batsman
np_players[skills == "Batsman"]
```

```
array([[ 74, 215],
       [ 73, 188],
       [ 69, 209],
       [ 71, 200],
       [ 76, 231],
       [ 73, 188],
       [ 73, 180],
       [ 70, 185],
       [ 79, 230],
       [ 72, 180],
       [ 71, 192],
       [ 75, 225],
       [ 77, 203],
       [ 73, 182],
       [ 75, 245],
       [ 74, 215],
       [ 71, 175],
       [ 73, 200],
       [ 74, 205],
       [ 76, 195]
```

In the snippet given above, we see that we can not only apply conditional subsetting on the same array but also use the condition on another array of the same size as that of our original array.

Creating NumPy Arrays (Of Fixed Length)

The following ways are commonly used to create arrays when you know the length of the arrays beforehand.

- np.ones(): It is used to create an array of 1s.
- np.zeros(): It is used to create an array of 0s.
- np.arange(): It is used to create an array with increments of fixed step size.
- np.linspace(): It is used to create an array of fixed length.

Let's take a look at the implementation of the functions given above in the following code snippet.

```
import numpy as np
```

```
np_1 = np.arange(1,5)
np_1
```

```
array([1, 2, 3, 4])
```

```
np_2 = np.zeros(5)
np_2, np_2.dtype
```

```
(array([0., 0., 0., 0., 0.]), dtype('float64'))
```

```
np.zeros(5, dtype = "int")
```

```
array([0, 0, 0, 0, 0])
```

When we try to add np_1 and np_2, it gives an error, as both are of different sizes.

```
np_1 + np_2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-41-79e2cc30c53b> in <module>
----> 1 np_1 + np_2

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

Therefore, we need to be careful about the sizes of NumPy arrays while using operations on them.

```
np_3 = np.ones(5)
np_3, np_3.size
```

```
(array([1., 1., 1., 1., 1.]), 5)
```

```
np_2 - np_3
```

```
array([-1., -1., -1., -1., -1.])
```

Linspace has the following syntax:

numpy.linspace(start, stop, num = 50, endpoint = True, retstep = False, dtype = None): Returns number spaces evenly with respect to interval. It is similar to 'arange', but instead of a step, it uses a sample number.

Parameters:

- **start:** [optional] Start of interval range. By default, start = 0.
- **stop:** End of interval range
- **restep:** If True, return (samples, step). By default, restep = False.
- **num:** [int, optional] Number of samples to be generated
- **dtype:** Type of output array

Return:

- **ndarray**
- **step:** [float, optional], if restep = True

But, we can use it as linspace(start, stop, samples):

```
np_4 = np.linspace(1, 10, 5)
np_4
```

```
array([ 1.  ,  3.25,  5.5 ,  7.75, 10.  ])
```

There are a few more useful functions for creating arrays as mentioned below.

- np.full(): It is used to create a constant array of any number 'n'.
- np.tile(): It is used to create a new array by repeating an existing array for a fixed number of times.
- np.eye(): It is used to create an identity matrix of any dimension.
- np.random.random(): It is used to create an array of random numbers between 0 and 1.
- np.random.randint(): It is used to create a random array of integers within a particular range.

Now, let's take a look at some other important functions that you can use while creating your arrays. These functions will help you modify the elements created using the commands mentioned above.

```
np_1
```

```
array([1, 2, 3, 4])
```

```
np.power(np_1, 2) #returns squares of all elements
```

```
array([ 1,  4,  9, 16], dtype=int32)
```

```
np.absolute(np.array([1, 2, -1, -2])) #returns positive value of al elements
```

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

The following functions will help you alter the arrays according to your requirements.

- np.power(): It calculates the powers of the array elements.
- np.absolute(): It converts all the elements in the absolute form.
- np.sin() or np.cos(): It takes the sine/cosine of the elements present in the array.
- np.log(): It takes the log of the elements in the array.

Another important feature offered by NumPy is **empty arrays**. You can initialise an empty array and later use it to store the output of your operations.

```
x = np.arange(1, 6)
y = np.empty(5)
np.multiply(x, 10, out = y) # this will store the output into empty array y
```

```
array([10., 20., 30., 40., 50.])
```

```
y
```

```
array([10., 20., 30., 40., 50.])
```

Once you have created an array, you may want to run aggregation operations over the data stored in it. An aggregation function helps you summarise the numerical data.

```
x = np.arange(1, 6)
x
```

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

```
np.add.reduce(x) # add all elements and return the sum
```

```
15
```

```
np.add.accumulate(x) # adds elements, saves it into running format and returns the cumulative table
```

```
array([ 1,  3,  6, 10, 15], dtype=int32)
```

Using the reduce() and accumulate() functions, you can easily summarise the data available in arrays. The reduce() function results in a single value, whereas the accumulate() function helps you apply your aggregation sequentially on each element of an array. These functions require a base function to aggregate the data, for example, add() in the case given above.

You can use the help() function to see all the possible functions available in its library.

```
help(np.linalg)
```

```
Help on package numpy.linalg in numpy:

NAME
    numpy.linalg

DESCRIPTION
    ``numpy.linalg``
    ================

    The NumPy linear algebra functions rely on BLAS and LAPACK to provide efficient
    low level implementations of standard linear algebra algorithms. Those
    libraries may be provided by NumPy itself using C versions of a subset of their
    reference implementations but, when possible, highly optimized libraries that
    take advantage of specialized processor functionality are preferred. Examples
    of such libraries are OpenBLAS, MKL (TM), and ATLAS. Because those libraries
    are multithreaded and processor dependent, environmental variables and external
    packages such as threadpoolctl may be needed to control the number of threads
    or specify the processor architecture.

    - OpenBLAS: https://www.openblas.net/
```

Manipulating NumPy Arrays

While working on NumPy arrays, you may have to manipulate data to obtain the desired results. NumPy arrays provide various features, which include changing the shape and combining and splitting arrays.

Stacking is performed using the np.hstack() and np.vstack() methods. For horizontal stacking, the number of rows should be the same, while for vertical stacking, the number of columns should be the same.

```
import numpy as np
arr_1 = np.arange(5)
arr_2 = np.arange(5, 10)
np.hstack((arr_1, arr_2)) #to horizontally stack both arrays(concatenate)
```

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

```
arr_3 = np.vstack((arr_1, arr_2)) #vertically stacked
arr_3
```

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

```
arr_3.shape
```

```
(2, 5)
```

Similar to stacking, another manipulation that is performed over arrays is reshaping. It is done using the reshape() function. It helps you change the dimensions of the existing arrays.

We can also use the reshape() function to manipulate the shape of an array.

```
arr_4 = np.arange(1, 11)
arr_4
```

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

```
arr_5 = arr_4.reshape(5, 2)
arr_5
```

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

```
arr_5.shape
```

```
(5, 2)
```

We also have to consider the shape of an array while using vstack() or hstack(); otherwise, these functions will give an error.

```
#for vstack axis=1 should have same value
arr_6 = np.arange(20,40).reshape(10,2)
arr_6
```

```
array([[20, 21],
       [22, 23],
       [24, 25],
       [26, 27],
       [28, 29],
       [30, 31],
       [32, 33],
       [34, 35],
       [36, 37],
       [38, 39]])
```

```
np.vstack((arr_5, arr_6))
```

```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [20, 21],
       [22, 23],
       [24, 25],
       [26, 27],
       [28, 29],
       [30, 31],
       [32, 33],
       [34, 35],
       [36, 37],
       [38, 39]])
```

When using hstack(), always check if for axis = 0 values for both the arrays, and for vstack(), check whether or not axis = 1 values for both the arrays are the same.

```
arr_6 = np.arange(20, 40).reshape(5, 4)
arr_6
```

```
array([[20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31],
       [32, 33, 34, 35],
       [36, 37, 38, 39]])
```

```
np.vstack((arr_5, arr_6)) # gives error as arr_5 is (5, 2) but arr-6 is (5,4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-76-757d875c3e4f> in <module>
----> 1 np.vstack((arr_5, arr_6)) # gives error as arr_5 is (5, 2) but arr-6 is (5,4)

<__array_function__ internals> in vstack(*args, **kwargs)

~\anaconda3\lib\site-packages\numpy\core\shape_base.py in vstack(tup)
    281     if not isinstance(arrs, list):
    282         arrs = [arrs]
--> 283     return _nx.concatenate(arrs, 0)
    284
    285

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array at i
ndex 0 has size 2 and the array at index 1 has size 4
```

Here is a cheat sheet for you to quickly refer to the commands and syntax.

➔ Summary

In this session, you learnt about the most important package for scientific computing in Python: **NumPy**. The various operations that you also learnt about include:

- Arrays, which are the basic data structure in the NumPy library
- How to create NumPy arrays from a list or a tuple
- How to create randomly large arrays using the arange command
- How to analyse the shape and dimension of an array using array.shape, array.ndim and so on
- Indexing, slicing and subsetting an array, which are similar to indexing in lists
- Operations on multidimensional arrays
- How to manipulate arrays using reshape(), hstack() and vstack()

The official documentation of Numpy is given in the link provided below.
NumPy User Guide

Matplotlib

Visualising data using plots and charts helps in gaining more insights into the data and also aids in presenting it. Graphics and visuals, if used intelligently and innovatively, can convey a lot more insights than what raw data alone can convey. Matplotlib serves the purpose of providing multiple functions to build graphs from the data stored in your lists, arrays, etc.

Facts and Dimensions

- Facts are numerical data.
- Dimensions are metadata, which is data that explains some other data, associated with fact variables.

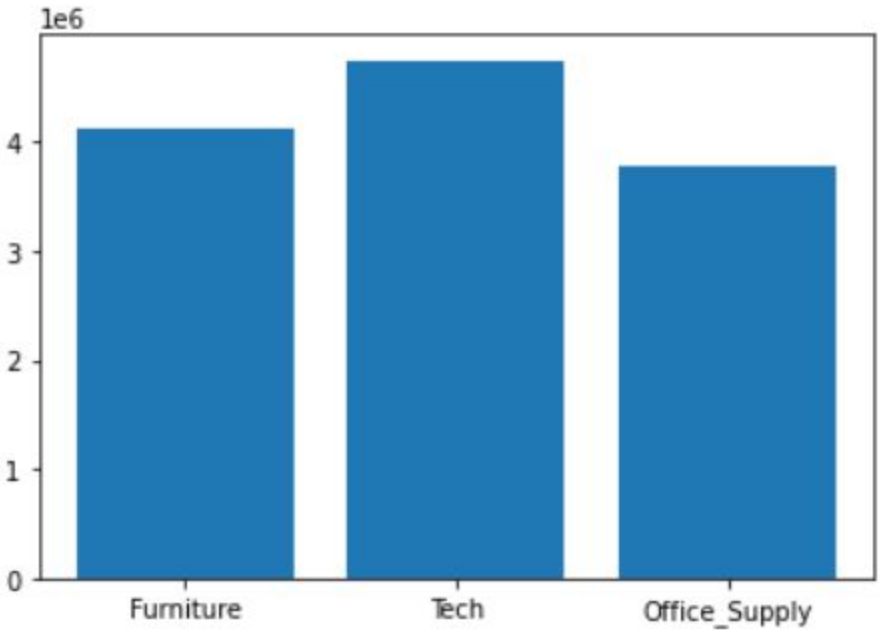
Bar Graph Visualisation [Link to data]

```
# importing numpy and the pyplot package of matplotlib
import numpy as np
import matplotlib.pyplot as plt
```

```
# Creating an array with product categories
product_category = np.array(['Furniture', 'Tech', 'Office_Supply'])
sales = np.array([4110451.90, 4744557.50, 3787492.52])
```

```
plt.bar(product_category, sales)
plt.show()
```

Output:



Input:

```
# plotting the bar graph with product categories on x-axis and sales amount
of y-axis
```

```
plt.bar(product_category, sales)
# adding title to the graph
plt.title("Sales across Product Categories", fontdict = {'fontsize': 20,
'fontweight': 5, 'color': 'Green'})
# labeling axes changing color of the bars in the bar graph
plt.xlabel("Product_Categories", fontdict = {'fontsize': 10, 'fontweight':
3, 'color': 'Blue'})
plt.ylabel("Sales", fontdict = {'fontsize': 10, 'fontweight': 3, 'color':
'Blue'})
# necessary command to display the created graph
plt.show()
```

```
tick_labels = ["0L", "10L", "20L", "30L", "40L", "50L", "60L", "70L"]
plt.yticks(tick_values, tick_labels)
```

Output:

Scatter plot Visualisation

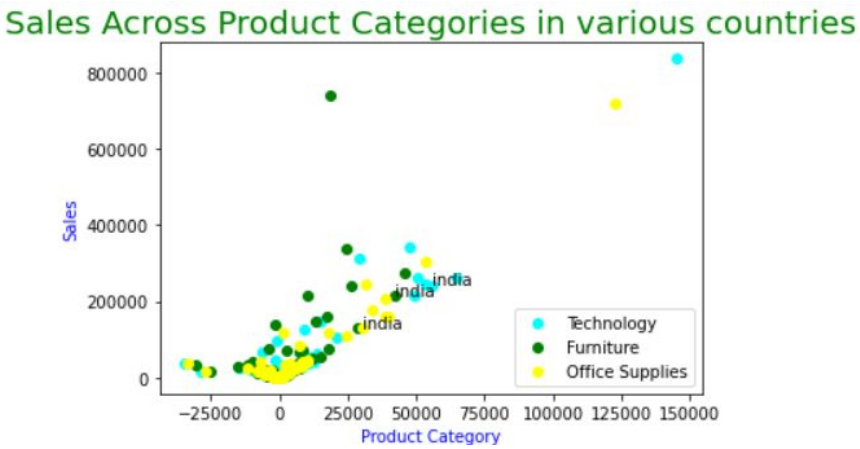
A scatter plot, as the name suggests, displays the scatter of the data. It can be helpful in checking for any relationship pattern between two quantitative variables and detecting the presence of outliers within them.
Matplotlib also offers a feature that allows incorporating a categorical distinction between the points plotted on a scatter plot. You can colour-code the points based on the category and distinguish them from each other.
Another feature of a scatter plot is that the points can be further distinguished over another dimension variable using labels. In the example we are using here, you have another array called 'country' that tells you the country where the sales were made. Suppose you want to highlight the points belonging to a particular country in the figure given above. Let's see how you can do that.

Input:

```
product_categories = np.array(["Technology", "Furniture", "Office
Supplies"])
colors = np.array(["cyan", "green", "yellow"])
# plotting the scatter plot with color coding the points belonging to
different categories
for color, category in zip(colors, product_categories):
    sales_category = sales[product_category == category]
    profit_category = profit[product_category == category]
    plt.scatter(profit_category, sales_category, c = color, label =
category)
# labeling points that belong to country "India"
for xy in zip(profit[country == "India"], sales[country == "India"]):
    plt.annotate(s = "india", xy = xy)
# Adding and formatting title
plt.title("Sales Across Product Categories in various countries",
fontdict={'fontsize': 20, 'fontweight' : 5, 'color' : 'Green'})
# Labeling Axes
plt.xlabel("Product Category", fontdict={'fontsize': 10, 'fontweight' : 3,
'color' : 'Blue'})
plt.ylabel("Sales", fontdict={'fontsize': 10, 'fontweight' : 3, 'color' :
'Blue'})
# Adding legend for interpretation of points
```

```
plt.legend()
plt.show()
```

Output:



Line Graph and Histogram

- Line Graph

A line graph is used to present continuous time-dependent data. It accurately depicts the trend of a variable over a specified time period.

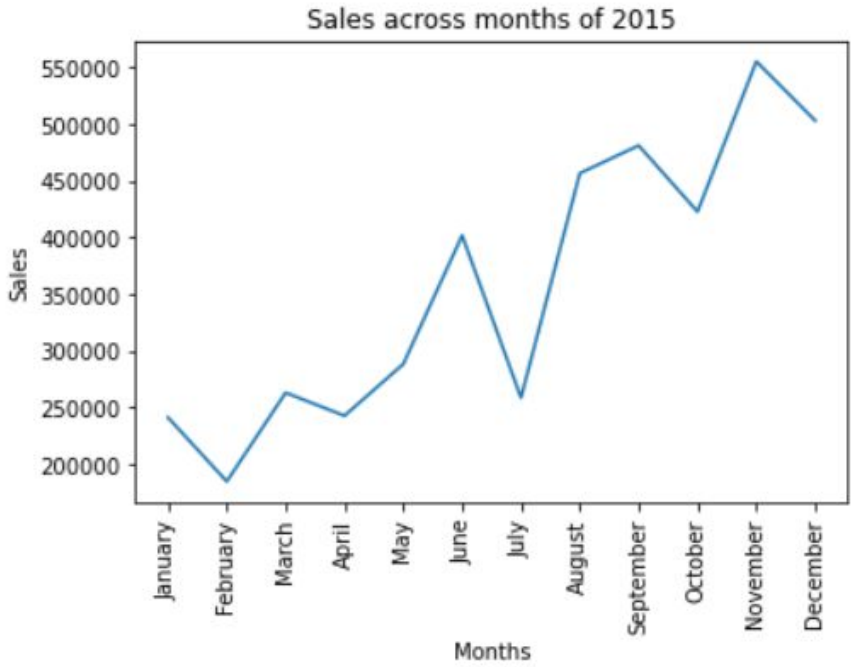
Input:

```
# importing the required libraries
import numpy as np
import matplotlib.pyplot as plt
# Sales data across months
months = np.array(['January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December'])
sales = np.array([241268.56, 184837.36, 263100.77, 242771.86, 288401.05,
401814.06, 258705.68, 456619.94, 481157.24, 422766.63, 555279.03,
503143.69])
```

```
# plotting a line chart
plt.plot(months, sales)
# adding title to the chart
```

```
plt.title("Sales across months of 2015")
# labeling the axes
plt.xlabel("Months")
plt.ylabel("Sales")
# rotating the tick values of x-axis
plt.xticks(rotation= 90)
# displaying the created plot
plt.show()
```

Output:



A line graph is extremely helpful when you want to understand the trend of a variable. Some key industries and services that rely on line graphs are financial markets, weather forecast, etc.

➔ Histogram

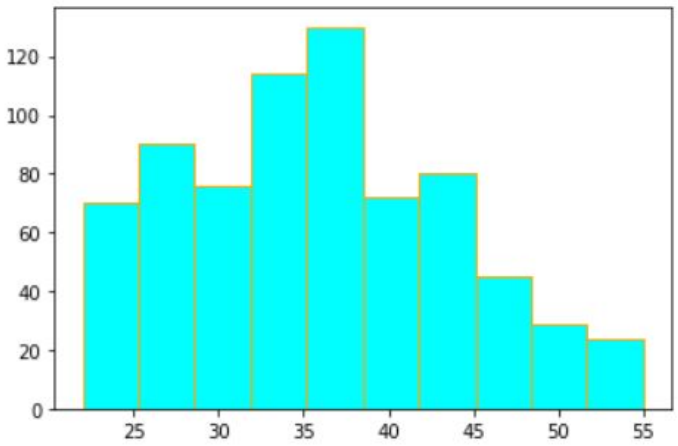
A histogram is a frequency chart that records the occurrence of an entry or element in a data set. It is useful when you want to understand the distribution of a given series.
Input:

```
# importing the required libraries - numpy, matplotlib.pyplot
import numpy as np
```

```
import matplotlib.pyplot as plt
# data corresponding to age of the employees in the company
age = np.array([23, 22, 24, 24, 23, 23, 22, 23, 24,......])
```

```
# plotting a histogram
plt.hist(age, bins = 10, edgecolor = "orange", color = "cyan")
plt.show()
```

Output:



Box Plot Visualisation

Box plots are extremely effective in summarising the spread of large data into a visual representation. They take the help of percentiles to divide a data range.
The percentile value gives the proportion of the data range that falls below a chosen data point when all the data points are arranged in descending order.
The figure given below summarises what a boxplot represents.

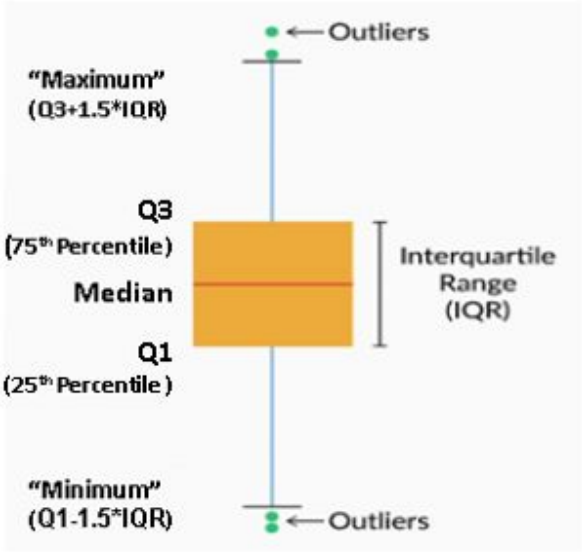


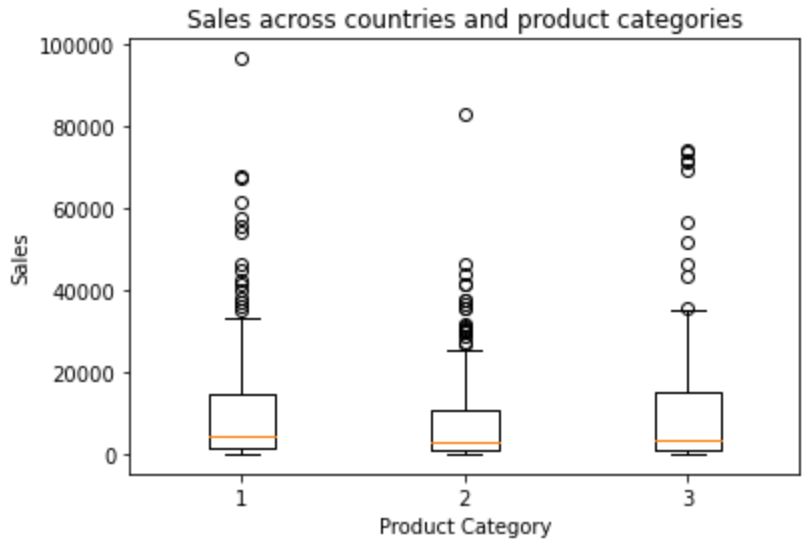
Fig: Boxplot

Input:

```
import numpy as np
import matplotlib.pyplot as plt
# Creating arrays with sales in different countries across each category:
'Furniture', 'Technology' and 'Office Supplies'
sales_technology = np.array ([1013.14, 8298.48, 875.51, 22320.83, ......])
sales_office_supplies = np.array ([1770.13, 7527.18, 1433.65, 423.3,
......])
sales_furniture = np.array ([981.84, 10209.84, 156.56, .....])
```

```
# plotting box plot for each category
plt.boxplot([sales_technology, sales_office_supplies, sales_furniture])
# adding title to the graph
plt.title("Sales across countries and product categories")
# labeling the axes
plt.xlabel("Product Category")
plt.ylabel("Sales")
# Replacing the x ticks with respective category
#plt.xticks((1,2,3), ["Technology","Office_supplies", "Furniture"])
plt.show()
```

Output:



Box plots divide the data range into three important categories, which are as follows:

- Median value: This is the value that divides the data range into two equal halves, that is, the 50th percentile.
- Interquartile range (IQR): These are data points between the 25th and 75th percentile values.
- Outliers: These are data points that differ significantly from other observations and lie beyond the whiskers.

Subplots

Sometimes, it is beneficial to draw different plots on a single grid next to each other in order to get a better overall view. Different plots presented in a single plot object are commonly referred to as subplots.

Input:

```
# importing numpy and the pyplot package of matplotlib
import numpy as np
import matplotlib.pyplot as plt# Data for sales made by the company
across three regions over the years 2012 to 2015
years = np.array(['2012', '2013', '2014', '2015'])
sales_Africa = np.array([127187.27, 144480.70, 229068.79, 283036.44])
sales_Asia_Pacific = np.array([713658.22, 863983.97, 1092231.65,
1372784.40])
```

```
sales_Europe = np.array([540750.63, 717611.40, 848670.24,
1180303.95])
```

Single Chart:

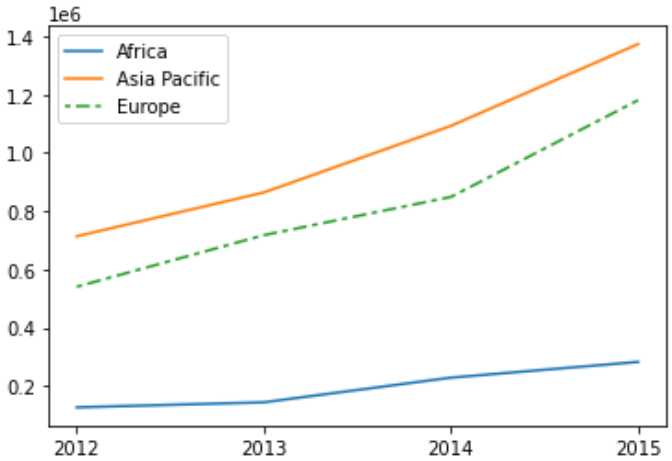
```
# Plotting the trend of sales in African region over the 4 years
# Check the array names before plotting to avoid error
fig, ax = plt.subplots() # It initiates a figure which will be used
to comprise multiple graphs in a single chart.
africa,= ax.plot(years, sales_Africa)
africa.set_label("Africa")

aspac,= ax.plot(years, sales_Asia_Pacific)
aspac.set_label("Asia Pacific")

europe,= ax.plot(years, sales_Europe)
europe.set_label("Europe")
europe.set_dashes([2, 2, 5, 2])

plt.legend()
plt.show()
```

Output:



Subplots are a good way to show multiple plots together for comparison. They provide the ability to create an array of plots, and you can have multiple charts next to each other such as the elements of an array as given below.

Multiple Charts:

Input:

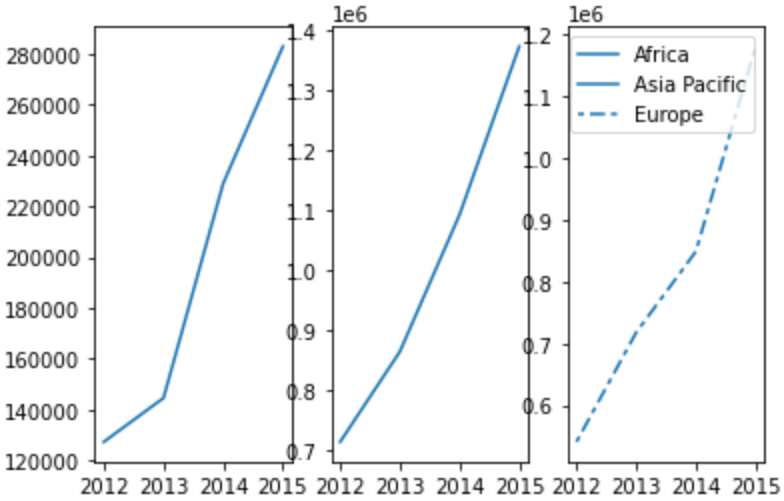
```
# Plotting the trend of sales in all three regions over the 4 years
using subplots
fig, ax = plt.subplots(ncols = 3, sharex = True)
africa,= ax[0].plot(years, sales_Africa)
africa.set_label("Africa")

aspac,= ax[1].plot(years, sales_Asia_Pacific)
aspac.set_label("Asia Pacific")

europe,= ax[2].plot(years, sales_Europe)
europe.set_label("Europe")
europe.set_dashes([2, 2, 5, 2])

plt.legend([africa, aspac, europe],  ["Africa", "Asia Pacific",
"Europe"])
plt.show()
```

Output:



Pandas Library

Pandas is a library built specifically for data analysis and created using NumPy. You will be using Pandas extensively for performing data manipulation or visualisation, building machine learning models, etc.

The two main data structures in Pandas are Series and DataFrames. The default way to store data is by using data frames; thus, manipulating data frames quickly is probably the most important skill for data analysis.

Pandas Series: It is similar to a 1-D NumPy array, and contains scalar values of the same type (numeric, character, date/time, etc.). A data frame is simply a table where each column is a Pandas series.

Creating a Pandas Series

You can create a Pandas series from array-like objects using pd.Series(data, dtype). Each element in the series has an index, which starts from 0.

```
# import pandas, pd is an alias
import pandas as pd
```

```
import numpy as np

# Creating a numeric pandas series
s = pd.Series([2, 4, 5, 6, 9])
print(s)
print(type(s))
```

Output:

```
0    2
1    4
2    5
3    6
4    9
dtype: int64
<class 'pandas.core.series.Series'>
```

Pandas DataFrame

A Pandas DataFrame is the most widely used data structure in data analysis. It is a table with rows and columns, where rows have indices and columns have meaningful names.

Creating Data Frames From Dictionaries

Various ways of creating data frames include using dictionaries, JSON objects and CSV files, and reading from text files.

```
# keys become column names
df = pd.DataFrame({'name': ['Vinay', 'Kushal', 'Aman', 'Saif'],
                   'age': [22, 25, 24, 28],
                    'occupation': ['engineer', 'doctor', 'data analyst',
'teacher']})
df
```

Output:

	name	age	occupation
0	Vinay	22	engineer
1	Kushal	25	doctor
2	Aman	24	data analyst
3	Saif	28	teacher

➔ Importing CSV Files as Pandas DataFrames

You can read CSV files as shown in the code snippet given below.

```
# reading a CSV file as a dataframe
# format: pd.read_csv(filepath, sep=',', header='infer')
market_df = pd.read_csv("global_sales_data/market_fact.csv")
market_df
```

Output:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38

➔ Sorting Data Frames

You can sort data frames in the following two ways:

1) By indices
2) By values

Firstly, you will sort DataFrames using indices as shown in the code snippet given below.

```
# Sorting by index
# axis = 0 indicates that you want to sort rows (use axis=1 for
columns)
market_df.sort_index(axis = 0, ascending = False).head()
```

Output:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
Ord_999	Prod_15	SHP_1383	Cust_361	5661.0800	0.00	33	1055.47	30.00	0.62
Ord_998	Prod_8	SHP_1380	Cust_372	750.6600	0.00	33	120.05	4.00	0.60
Ord_998	Prod_5	SHP_1382	Cust_372	2149.3700	0.03	42	217.87	19.99	0.55
Ord_998	Prod_8	SHP_1381	Cust_372	254.3200	0.01	8	-117.39	6.50	0.79
Ord_997	Prod_14	SHP_1379	Cust_365	28761.5200	0.04	8	285.11	24.49	0.37

Secondly, you will sort DataFrames using values as shown in the code snippet given below.

```
# Sorting by values

# Sorting in decreasing order of Sales we use an attribute called
ascending which is boolean for descending we use false and true for
ascending.
market_df.sort_values(by='Sales',ascending=false).head()
```

Output:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
Ord_3084	Prod_17	SHP_4279	Cust_1151	89061.05	0.00	13	27220.69	24.49	0.39
Ord_2338	Prod_17	SHP_3207	Cust_932	45923.76	0.07	7	102.61	24.49	0.39
Ord_3875	Prod_17	SHP_5370	Cust_1351	41343.21	0.09	8	3852.19	24.49	0.39
Ord_2373	Prod_14	SHP_3259	Cust_942	33367.85	0.01	9	3992.52	24.49	0.37
Ord_4614	Prod_14	SHP_6423	Cust_1571	29884.60	0.05	49	12748.86	24.49	0.44

➔ Setting Index Functionality

Data:

	1|Akshay|Mathematics|50|40|80
0	2|Mahima|English|40|33|83
1	3|Vikas|Mathematics|50|42|84
2	4|Abhinav|English|40|31|78
3	5|Mahima|Science|50|40|80
4	6|Akshay|Science|50|49|98

Input:

```
df = pd.read_csv("Downloads\marks.csv", header = None, sep = '|',
index_col = 0)
```

Output:

S.No.	Name	Subject	Maximum Marks	Marks Obtained	Percentage
1	Akshay	Mathematics	50	40	80
2	Mahima	English	40	33	83
3	Vikas	Mathematics	50	42	84
4	Abhinav	English	40	31	78
5	Mahima	Science	50	40	80

Describing Data

➔ Head Function

Input:

```
import pandas as pd
import numpy as np
df = pd.read_excel("Downloads\sales.xlsx")
df.head() #Displays 5 top rows by default and x number of entries as passed
as the parameter.
```

Output:

	Market	Region	No_of_Orders	Profit	Sales
0	Africa	Western Africa	251	-12901.51	78476.06
1	Africa	Southern Africa	85	11768.58	51319.50
2	Africa	North Africa	182	21643.08	86698.89
3	Africa	Eastern Africa	110	8013.04	44182.60
4	Africa	Central Africa	103	15606.30	61689.99

Input:

```
df.describe()
# Describe gives you a summary of all the numeric columns in the dataset df
Similarly, to display the last entries, you can use the tail()
```

Output:

	No_of_Orders	Profit	Sales
count	23.000000	23.000000	23.000000
mean	366.478261	28859.944783	206285.108696
std	246.590361	27701.193773	160589.886606
min	37.000000	-16766.900000	8190.740000
25%	211.500000	12073.085000	82587.475000
50%	356.000000	20948.840000	170416.310000
75%	479.500000	45882.845000	290182.375000
max	964.000000	82091.270000	656637.140000

Indexing and Slicing Data

➔ Selecting Rows

Selecting rows in data frames is similar to indexing in NumPy arrays, which you already looked at. The syntax df[start_index:end_index] will subset rows according to the start and end indices. You can select the rows from indices 2 to 6 as shown in the code snippet given below.

```
import numpy as np
import pandas as pd

market_df = pd.read_csv("../global_sales_data/market_fact.csv")
# Selecting the rows from indices 2 to 6
market_df[2:7]
```

Output:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	6.15	0.37
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	0.99	0.55
9	Ord_4725	Prod_6	SHP_6593	Cust_1641	57.2200	0.07	8	-27.72	6.60	0.37
11	Ord_1925	Prod_6	SHP_2637	Cust_708	465.9000	0.05	38	79.34	4.86	0.38
13	Ord_2207	Prod_11	SHP_3093	Cust_839	3364.2480	0.10	15	-693.23	61.76	0.78

➔ Selecting Columns

The two simple ways to select a single column from a dataframe are as follows:

- df['column_nam df['column_name']
- df.column_name

```
# Using df['column']
sales = market_df['Sales']
sales.head()
```

Output:

```
0     136.81
1      42.27
2    4701.69
3    2337.89
4    4233.15
Name: Sales, dtype: float64
```

➔ Selecting Columns

You can select multiple columns by passing the list of column names inside the [ ]: df[['column_1', 'column_2', 'column_n']].

```
# Select Cust_id, Sales and Profit:
market_df[['Cust_id', 'Sales', 'Profit']].head()
```

Output:

	Cust_id	Sales	Profit
0	Cust_1818	136.81	-30.51
1	Cust_1818	42.27	4.56
2	Cust_1818	4701.69	1148.90
3	Cust_1818	2337.89	729.34
4	Cust_1818	4233.15	1219.87

Input:

```
df = pd.read_excel("Downloads\sales.xlsx" , index_col = (0,1))
df.loc[["Africa", "Europe"], ["Profit", "Sales"]]
# dataframe.loc[[list_of_row_labels], [list_of_column_labels]] : You
can use the loc method to extract rows and columns from a dataframe
based on the following labels.This is called label-based indexing
over dataframes
```

Output:

Market	Region	Profit	Sales
Africa	Western Africa	-12901.51	78476.06
	Southern Africa	11768.58	51319.50
	North Africa	21643.08	86698.89
	Eastern Africa	8013.04	44182.60
	Central Africa	15606.30	61689.99
Europe	Western Europe	82091.27	656637.14
	Southern Europe	18911.49	215703.93
	Northern Europe	43237.44	252969.09
	Eastern Europe	25050.69	108258.93

Input:

```
df.iloc[0:3, 0:2] #Since we use positions instead of labels to
extract values from the dataframe, it is called position-based
indexing. With these two methods, you can easily extract the required
entries from a dataframe based on their labels or positions
```

Output:

:

Market	Region	No_of_Orders	Profit
Africa	Western Africa	251	-12901.51
	Southern Africa	85	11768.58
	North Africa	182	21643.08

Input:

```
df[(df["Profit"]>0)].head() #Often, you want to select rows that
```

```
satisfy some given conditions
```

Output:

Market	Region	No_of_Orders	Profit	Sales
Africa	Southern Africa	85	11768.58	51319.50
	North Africa	182	21643.08	86698.89
	Eastern Africa	110	8013.04	44182.60
	Central Africa	103	15606.30	61689.99
Asia Pacific	Southern Asia	469	67998.76	351806.60

➔ Grouping and Pivoting Data Frames

Grouping and aggregation are some of the most frequently used operations in data analysis, especially while performing exploratory data analysis (EDA), where comparing summary statistics across groups of data is common.
For example, in the retail sales data that you are working with, you may want to compare the average sales of various regions or the total profit of two customer segments.
Grouping analysis can consist of the following three parts:

1. Splitting the data into groups (For example, groups of customer segments and product categories)
2. Applying a function to each group (For example, mean or total sales of each customer segment)
3. Combining the results into a data structure showing the summary statistics

First, we will merge all the DataFrames, so that we have all the data in one master_df as shown in the code snippet given below.

```
# Loading libraries and files
import numpy as np
import pandas as pd
market_df = pd.read_csv("../global_sales_data/market_fact.csv")
customer_df = pd.read_csv("../global_sales_data/cust_dimen.csv")
product_df = pd.read_csv("../global_sales_data/prod_dimen.csv")
shipping_df = pd.read_csv("../global_sales_data/shipping_dimen.csv")
orders_df = pd.read_csv("../global_sales_data/orders_dimen.csv")
# Merging the DataFrames one by one
df_1 = pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df_2 = pd.merge(df_1, product_df, how='inner', on='Prod_id')
df_3 = pd.merge(df_2, shipping_df, how='inner', on='Ship_id')
master_df = pd.merge(df_3, orders_df, how='inner', on='Ord_id')
master_df.head()
```

Output:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	...	Region	Customer_Segment
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	...	WEST	CORPORATE
1	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	...	WEST	CORPORATE
2	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64	6.15	0.37	...	WEST	CORPORATE
3	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12	3.37	0.57	...	ONTARIO	HOME OFFICE
4	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58	3.98	0.56	...	WEST	CONSUMER

5 rows × 22 columns

_Margin	...	Region	Customer_Segment	Product_Category	Product_Sub_Category	Order_ID_x	Ship_Mode	Ship_Date	Order_ID_y	Order_Date	Order_Priority
0.56	...	WEST	CORPORATE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	36262	REGULAR AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED
0.59	...	WEST	CORPORATE	TECHNOLOGY	TELEPHONES AND COMMUNICATION	36262	EXPRESS AIR	27-07-2010	36262	27-07-2010	NOT SPECIFIED
0.37	...	WEST	CORPORATE	OFFICE SUPPLIES	PAPER	36262	EXPRESS AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED
0.57	...	ONTARIO	HOME OFFICE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	37863	REGULAR AIR	26-02-2011	37863	24-02-2011	HIGH
0.56	...	WEST	CONSUMER	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	53026	REGULAR AIR	03-03-2012	53026	26-02-2012	LOW

Step 1: Grouping Using df.groupby()

```
# Which customer segments are the least profitable?
# Step 1. Grouping: First, we will group the dataframe by customer segments
df_by_segment = master_df.groupby('Customer_Segment')
```

```
df_by_segment
```

Output:

```
<pandas.core.groupby.DataFrameGroupBy object at 0x1046be710>
```

Step 2: Applying a Function

```
# We can choose aggregate functions such as sum, mean, median, etc.
df_by_segment['Profit'].sum()
```

Output:

```
Customer_Segment
CONSUMER          287959.94
CORPORATE         599746.00
HOME OFFICE       318354.03
SMALL BUSINESS    315708.01
Name: Profit, dtype: float64
```

Step 3: Combining Results Into Data Structures

We will combine the results in a data structure as shown in the code snippet given below.

```
# Converting to a df
pd.DataFrame(df_by_segment['Profit'].sum())
# Let's go through some more examples
# E.g.: Which product categories are the least profitable?

# 1. Group by product category
by_product_cat = master_df.groupby('Product_Category')
# E.g.: Which product categories and sub-categories are the least
profitable?
# 1. Group by category and sub-category
by_product_cat_subcat = master_df.groupby(['Product_Category',
'Product_Sub_Category'])
by_product_cat_subcat['Profit'].mean()
```

```
# E.g.: Which product categories and sub-categories are the least profitable?
# 1. Group by category and sub-category
by_product_cat_subcat = master_df.groupby(['Product_Category', 'Product_Sub_Category'])
by_product_cat_subcat['Profit'].mean()
```

Output:

```
Product_Category  Product_Sub_Category
FURNITURE         BOOKCASES                      -177.683228
                  CHAIRS & CHAIRMATS              387.693601
                  OFFICE FURNISHINGS              127.446612
                  TABLES                         -274.411357
OFFICE SUPPLIES   APPLIANCES                      223.866498
                  BINDERS AND BINDER ACCESSORIES  335.970918
                  ENVELOPES                       195.864228
                  LABELS                           47.490174
                  PAPER                            36.949551
                  PENS & ART SUPPLIES              11.950679
                  RUBBER BANDS                     -0.573575
                  SCISSORS, RULERS AND TRIMMERS   -54.161458
                  STORAGE & ORGANIZATION           12.205403
TECHNOLOGY        COMPUTER PERIPHERALS            124.389815
                  COPIERS AND FAX                1923.695287
                  OFFICE MACHINES                 913.094748
                  TELEPHONES AND COMMUNICATION    358.948607
Name: Profit, dtype: float64
```

```
df.pivot(columns='grouping_variable_col', values='value_to_aggregate', index='grouping_variable_row') #to pivot
```

Pivot Tables

You can use Pandas pivot tables as an alternative to groupby(). They provide Excel-like functionalities to create aggregate tables.
The general syntax is pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', ...)
where,

- ‘data’ is the data frame,
- ‘values’ contain the columns to aggregate within the dataset,
- ‘index’ is the row in the pivot table,
- ‘columns’ contain the columns that you want in the pivot table, and
- ‘aggfunc’ is the aggregate function.

Now, let's take a look at the data under the Product_Category column as given below.

```
master_df.pivot_table(columns = 'Product_Category')
```

Output:

	Sales
Customer_Segment	
CONSUMER	1857.859965
CORPORATE	1787.680389
HOME OFFICE	1754.312931
SMALL BUSINESS	1698.124841

We will compute the mean of numeric columns across categories as shown in the code snippet given below.

```
# Computes the mean of all numeric columns across categories
# Notice that the means of Order_IDs are meaningless
master_df.pivot_table(columns = 'Product_Category')
```

Output:

Product_Category	FURNITURE	OFFICE SUPPLIES	TECHNOLOGY
Discount	0.049287	0.050230	0.048746
Order_ID_x	30128.711717	30128.122560	29464.891525
Order_ID_y	30128.711717	30128.122560	29464.891525
Order_Quantity	25.709977	25.656833	25.266344
Product_Base_Margin	0.598555	0.461270	0.556305
Profit	68.116531	112.369544	429.208668
Sales	3003.822820	814.048178	2897.941008
Shipping_Cost	30.883811	7.829829	8.954886
is_profitable	0.465197	0.466161	0.573366
profit_per_qty	-3.607020	1.736175	-52.274216

Operations on DataFrames

Input:

```
import pandas as pd
import numpy as np
df = pd.read_excel("Downloads\sales.xlsx" , index_col = 1)
df.rename(columns={"Market" : "Country"}).head()
```

Output:

Region	Country	No_of_Orders	Profit	Sales
Western Africa	Africa	251	-12901.51	78476.06
Southern Africa	Africa	85	11768.58	51319.50
North Africa	Africa	182	21643.08	86698.89
Eastern Africa	Africa	110	8013.04	44182.60
Central Africa	Africa	103	15606.30	61689.99

➔ Lambda Function

Input:

```
df["Profit"] = df.Profit.apply(lambda x: np.nan if x<0 else x)
df.head()
```

Output:

Region	Market	No_of_Orders	Profit	Sales
Western Africa	Africa	251	NaN	78476.06
Southern Africa	Africa	85	11768.58	51319.50
North Africa	Africa	182	21643.08	86698.89
Eastern Africa	Africa	110	8013.04	44182.60
Central Africa	Africa	103	15606.30	61689.99

➔ Hierarchical Indexing

Input:

```
df.reset_index(inplace = True)
```

```
df.set_index(["Market", "Region"], inplace = True)
df.head()
#You have learnt how to create multilevel indices in the earlier segment
while loading data in Pandas. However, you can also alter the index column
after loading the data into a dataframe.
```

Output:

Market	Region	No_of_Orders	Profit	Sales
Africa	Western Africa	251	NaN	78476.06
	Southern Africa	85	11768.58	51319.50
	North Africa	182	21643.08	86698.89
	Eastern Africa	110	8013.04	44182.60
	Central Africa	103	15606.30	61689.99

Merge and Append

In this section, you will merge and concatenate multiple DataFrames. Merging is one of the most common operations that you will perform since data often comes in various files. The functions used for merge and append are as follows:

- Merge multiple data frames using common columns/keys and pd.merge()
- Concatenate data frames using pd.concat()

Now, we will merge DataFrames using the pd.merge() method as shown below.

```
# loading libraries and reading the data
import numpy as np
import pandas as pd

market_df = pd.read_csv("./global_sales_data/market_fact.csv")
customer_df = pd.read_csv("./global_sales_data/cust_dimen.csv")
product_df = pd.read_csv("./global_sales_data/prod_dimen.csv")
shipping_df = pd.read_csv("./global_sales_data/shipping_dimen.csv")
orders_df = pd.read_csv("./global_sales_data/orders_dimen.csv")
# Merging the dataframes
```

```
# Note that Cust_id is the common column/key, which is provided to the 'on' argument
# how = 'inner' makes sure that only the customer ids present in both dfs are included in the result
df_1 = pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df_1.head()
```

Output:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	Customer_Name	Province
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	AARON BERGMAN	ALBERTA
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54	AARON BERGMAN	ALBERTA
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	AARON BERGMAN	ALBERTA
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37	AARON BERGMAN	ALBERTA
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38	AARON BERGMAN	ALBERTA

➔ Concatenating DataFrames

Concatenation is much more straightforward than merging. It is used when you have dataframes with the same columns and want to append them (pile one on top of the other), or when you have the same rows and want to append them side by side.

```
# dataframes having the same columns
df1 = pd.DataFrame({'Name': ['Aman', 'Joy', 'Rashmi', 'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                   )

df2 = pd.DataFrame({'Name': ['Akhil', 'Asha', 'Preeti'],
                    'Age': ['31', '22', '23'],
                    'Gender': ['M', 'F', 'F']}
                   )
# To concatenate them, one on top of the other, you can use pd.concat
# The first argument is a sequence (list) of dataframes
# axis = 0 indicates that we want to concat along the row axis
pd.concat([df1, df2], axis = 0)
```

Output:

	Name	Age	Gender
0	Aman	34	M
1	Joy	31	M
2	Rashmi	22	F
3	Saif	33	M
0	Akhil	31	M
1	Asha	22	F
2	Preeti	23	F

➔ Concatenating DataFrames Having the Same Rows

You may also have dataframes with the same rows but different columns (and no common columns). In this case, you may want to concatenate them side by side.
We will concatenate the two DataFrames using the method shown in the following code snippet.

```
df1 = pd.DataFrame({'Name': ['Aman', 'Joy', 'Rashmi', 'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                   )
df2 = pd.DataFrame({'School': ['RK Public', 'JSP', 'Carmel
Convent', 'St. Paul'],
                    'Graduation Marks': ['84', '89', '76',
'91']}
                   )
# To join the two dataframes, use axis = 1 to indicate joining
along the columns axis
# The join is possible because the corresponding rows have the
same indices
pd.concat([df1, df2], axis = 1)
```

Output:

	Name	Age	Gender	School	Graduation Marks
0	Aman	34	M	RK Public	84
1	Joy	31	M	JSP	89
2	Rashmi	22	F	Carmel Convent	76
3	Saif	33	M	St. Paul	91

➔ Performing Arithmetic Operations on More Than One Data Frame

Now, we will perform some operations on DataFrames.

```
# Teamwise stats for IPL 2018
IPL_2018 = pd.DataFrame({'IPL Team': ['CSK', 'SRH', 'KKR', 'RR',
'MI', 'RCB', 'KXIP', 'DD'],
                         'Matches Played': [16, 17, 16, 15, 14, 14,
14, 14],
                         'Matches Won': [11, 10, 9, 7, 6, 6, 6, 5]}
                        )
# Set the 'IPL Team' column as the index to perform arithmetic
operations on the other rows using the team as reference
IPL_2018.set_index('IPL Team', inplace = True)
# Similarly, we have the stats for IPL 2017
IPL_2017 = pd.DataFrame({'IPL Team': ['MI', 'RPS', 'KKR', 'SRH',
'KXIP', 'DD', 'GL', 'RCB'],
                         'Matches Played': [17, 16, 16, 15, 14, 14,
14, 14],
                         'Matches Won': [12, 10, 9, 8, 7, 6, 4, 3]}
                        )
IPL_2017.set_index('IPL Team', inplace = True)
# Simply add the two DFs using the add opearator
Total = IPL_2018 + IPL_2017
Total
```

Output:

IPL Team	Matches Played	Matches Won
CSK	NaN	NaN
DD	28.0	11.0
GL	NaN	NaN
KKR	32.0	18.0
KXIP	28.0	13.0
MI	31.0	18.0
RCB	28.0	9.0
RPS	NaN	NaN
RR	NaN	NaN
SRH	32.0	18.0

You might have noticed that there are a lot of Not a Number (NaN) values. This is because some teams that played in IPL 2017 were not present in IPL 2018. In addition, there were new teams present in IPL 2018. We can handle these NaN values using df.add() instead of the simple add operator. Let's understand how.

Handling Time-Series Data

DataFrame:
Input:

```
import pandas as pd
import numpy as np
df = pd.read_csv("Downloads\weather_data.csv")
df.head()
```

Output:

	Temperature	DewPoint	Pressure	Date_Time
0	46.2	37.5	1	20100101 00:00
1	44.6	37.1	1	20100101 01:00
2	44.1	36.9	1	20100101 02:00
3	43.8	36.9	1	20100101 03:00
4	43.5	36.8	1	20100101 04:00

Parsing date and time:

Input:

```
df = pd.read_csv("Downloads\weather_data.csv", parse_dates =
["Date_Time"])
df.head()
```

Output:

	Temperature	DewPoint	Pressure	Date_Time
0	46.2	37.5	1	2010-01-01 00:00:00
1	44.6	37.1	1	2010-01-01 01:00:00
2	44.1	36.9	1	2010-01-01 02:00:00
3	43.8	36.9	1	2010-01-01 03:00:00
4	43.5	36.8	1	2010-01-01 04:00:00

Input:

```
df = pd.read_csv("Downloads\weather_data.csv", parse_dates = True,
index_col = "Date_Time")
df.loc["February 5th, 2010"].head()
```

Output:

Date_Time	Temperature	DewPoint	Pressure
2010-02-05 00:00:00	48.2	40.9	1
2010-02-05 01:00:00	47.2	41.0	1
2010-02-05 02:00:00	46.5	40.6	1
2010-02-05 03:00:00	45.9	40.4	1
2010-02-05 04:00:00	45.4	39.9	1