# Activity

**Aim:** Python lambda function operations on series or data frames

**Learning outcome**:

**Duration**: 3 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

**Lambda Function**

Lambda function contains a single expression.

The Lambda function is a small function that can also use as an anonymous function means it doesn’t require any name. The lambda function is useful to solve small problems with less code.

The following syntax is used to apply a lambda function on pandas dataframe:

*dataframe.apply(lambda x: x+2)*

Applying Lambda Function on a Single Column Using DataFrame.assign() Method

The dataframe.assign() method applies the Lambda function on a single column.

We have applied a lambda function on the column Students Marks. After applying the Lambda function, the student percentages are calculated and stored in a new Percentage column.

The following implementation applies a lambda function on a single column in Pandas dataframe.

import pandas as pd

# initialization of list

students\_record= [['Samreena',900],['Mehwish',750],['Asif',895],

['Mirha',800],['Affan',850],['Raees',950]]

# pandas dataframe creation

dataframe = pd.DataFrame(students\_record,columns=['Student Names','Student Marks'])

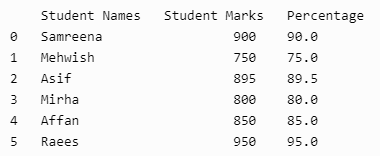
# using Lambda function

dataframe1 = dataframe.assign(Percentage = lambda x: (x['Student Marks'] /1000 \* 100))

# display dataframe

print(dataframe1)

## Output/Results snippet:



**Program:**

**Applying Lambda Function on Multiple Columns Using DataFrame.assign() Method**

We have four columns Student Names, Computer, Math, and Physics. We applied a Lambda function on multiple subjects columns such as Computer, Math, and Physics to calculate the obtained marks stored in the Marks\_Obtained column.

import pandas as pd

# nested list initialization

values\_list = [['Samreena',85, 75, 100], ['Mehwish', 90, 75, 90], ['Asif', 95, 82, 80],

['Mirha', 75, 88, 68], ['Affan', 80, 63, 70], ['Raees', 91, 64, 90]]

# pandas dataframe creation

df = pd.DataFrame(values\_list, columns=['Student Names','Computer', 'Math', 'Physics'])

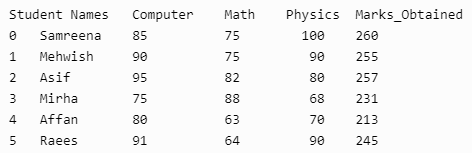
# applying Lambda function

dataframe = df.assign(Marks\_Obtained=lambda x: (x['Computer'] + x['Math'] + x['Physics']))

# display dataframe

print(dataframe)

## Output/Results snippet:



**Program:**

**Applying Lambda Function on a Single Row Using DataFrame.apply() Method**

The dataframe.apply() method applies the Lambda function on a single row.

We applied the lambda function a single row axis=1. Using the lambda function, we incremented each person’s Monthly Income by 1000.

import pandas as pd

df=pd.DataFrame({

'ID':[1,2,3,4,5],

'Names':['Samreena','Asif','Mirha','Affan','Mahwish'],

'Age':[20,25,15,10,30],

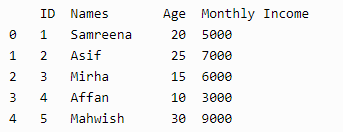
'Monthly Income':[4000,6000,5000,2000,8000]

})

df['Monthly Income']=df.apply(lambda x: x['Monthly Income']+1000,axis=1)

print(df)

## Output/Results snippet:



**Program:**

**Filtering Data by Applying Lambda Function**

We can also filter the desired data by applying the Lambda function.

The filter() function takes pandas series and a lambda function. The Lambda function applies to the pandas series that returns the specific results after filtering the given series.

We have applied the lambda function on the Age column and filtered the age of people under 25 years.

import pandas as pd

df=pd.DataFrame({

'ID':[1,2,3,4,5],

'Names':['Samreena','Asif','Mirha','Affan','Mahwish'],

'Age':[20,25,15,10,30],

'Monthly Income':[4000,6000,5000,2000,8000]

})

print(list(filter(lambda x: x<25,df['Age'])))

## Output/Results snippet:



**Program:**

**Use the map() Function by Applying Lambda Function**

We can use the map() and lambda functions.

The lambda function applies on series to map the series based on the input correspondence. This function is useful to substitute or replace a series with other values.

When we use the map() function, the input size will equal the output size.

import pandas as pd

df=pd.DataFrame({

'ID':[1,2,3,4,5],

'Names':['Samreena','Asif','Mirha','Affan','Mahwish'],

'Age':[20,25,15,10,30],

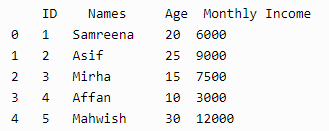
'Monthly Income':[4000,6000,5000,2000,8000]

})

df['Monthly Income']=list(map(lambda x: int(x+x\*0.5),df['Monthly Income']))

print(df)

## Output/Results snippet:



**Program:**

**Use if-else Statement by Applying Lambda Function**

We can also apply the conditional statements on pandas dataframes using the lambda function.

We used the conditional statement inside the lambda function. We applied the condition on the Monthly Income column.

If the monthly income is greater and equal to 5000, add Stable inside the Category column; otherwise, add Unstable.

import pandas as pd

df=pd.DataFrame({

'ID':[1,2,3,4,5],

'Names':['Samreena','Asif','Mirha','Affan','Mahwish'],

'Age':[20,25,15,10,30],

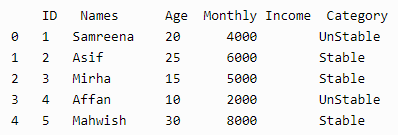
'Monthly Income':[4000,6000,5000,2000,8000]

})

df['Category']=df['Monthly Income'].apply(lambda x: 'Stable' if x>=5000 else 'UnStable')

print(df)

## Output/Results snippet:



**References:**

1. <https://www.delftstack.com/howto/python-pandas/apply-lambda-functions-to-pandas-dataframe/>

# Activity

**Aim:** Dealing with missing and noisy data

**Learning outcome**:

**Duration**: 3 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

## Dealing with Missing data

## import pandas as pd

## # Creating the dataframe

## df = pd.DataFrame({'Job Position': ['CEO', 'Senior Manager', 'Junior Manager', 'Employee', 'Assistant Staff'], 'Years of Experience':[5, 4, 3, None, 1], 'Salary':[100000,80000,None,40000, 20000]})

## # Viewing the contents of the dataframe

## df.head()

## Output/Results snippet:

## 

## Some of the ways to handle missing data are listed below:

## 1. Data Removal

## Remove the missing data rows (data points) from the dataset. However, when using this technique will decrease the available dataset and in turn result in less robustness of data point if the size of dataset is originally small.

## # Dropping the 2nd and 3rd index

## dropped\_df = df.drop([2,3],axis=0)

## # Viewing the dataframe

## dropped\_df

## Output/Results snippet:

## 

## 2. Fill missing value through statistical imputation

## Fill the missing data by taking the mean or median of the available data points. Generally, the median of the data points is used to fill the missing values as it is not affected heavily by outliers like the mean. Here, we have used the median to fill the missing data.

## # Filling each column with their mean values

## df['Years of Experience'] = df['Years of Experience'].fillna(df['Years of Experience'].mean())

## df['Salary'] = df['Salary'].fillna(df['Salary'].mean())

## # Viewing the dataframe

## df

## Output/Results snippet:

## 

## Program:

**Dealing with Noisy data**

Binning method is used to smoothing data or to handle noisy data. In this method, the data is first sorted and then the sorted values are distributed into a number of buckets or bins. As binning methods consult the neighborhood of values, they perform local smoothing.

There are three approaches to perform smoothing –

**Smoothing by bin means :** In smoothing by bin means, each value in a bin is replaced by the mean value of the bin.

**Smoothing by bin median :** In this method each bin value is replaced by its bin median value.

**Smoothing by bin boundary :** In smoothing by bin boundaries, the minimum and maximum values in a given bin are identified as the bin boundaries. Each bin value is then replaced by the closest boundary value.

Approach:

1. Sort the array of given data set.
2. Divides the range into N intervals, each containing the approximately same number of samples(Equal-depth partitioning).
3. Store mean/ median/ boundaries in each row.

Examples:

Sorted data for price (in dollars): 4, 8, 9, 15, 21, 21, 24, 25, 26, 28, 29, 34

Smoothing by bin means:

- Bin 1: 9, 9, 9, 9

- Bin 2: 23, 23, 23, 23

- Bin 3: 29, 29, 29, 29

Smoothing by bin boundaries:

- Bin 1: 4, 4, 4, 15

- Bin 2: 21, 21, 25, 25

- Bin 3: 26, 26, 26, 34

Smoothing by bin median:

- Bin 1: 9 9, 9, 9

- Bin 2: 24, 24, 24, 24

- Bin 3: 29, 29, 29, 29

Use the following basic syntax to perform data binning on a pandas DataFrame:

import pandas as pd

#perform binning with 3 bins

df['new\_bin'] = pd.qcut(df['variable\_name'], q=3)

Example 1

import pandas as pd

#create DataFrame

df = pd.DataFrame({'points': [4, 4, 7, 8, 12, 13, 15, 18, 22, 23, 23, 25],

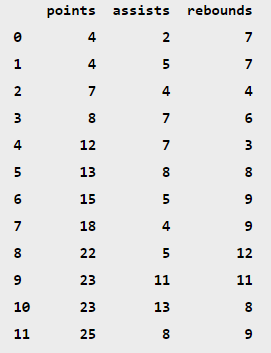
'assists': [2, 5, 4, 7, 7, 8, 5, 4, 5, 11, 13, 8],

'rebounds': [7, 7, 4, 6, 3, 8, 9, 9, 12, 11, 8, 9]})

#view DataFrame

print(df)

## Output/Results snippet:



**Perform Basic Data Binning**

The following code shows how to perform data binning on the points variable using the qcut() function with specific break marks:

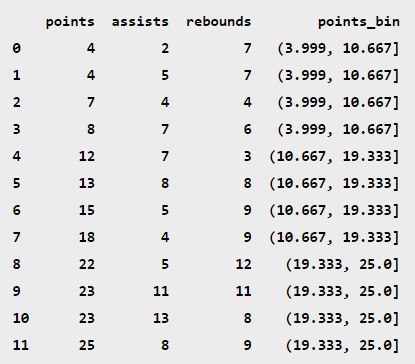
#perform data binning on points variable

df['points\_bin'] = pd.qcut(df['points'], q=3)

#view updated DataFrame

print(df)

## Output/Results snippet:



Notice that each row of the data frame has been placed in one of three bins based on the value in the points column.

We can use the value\_counts() function to find how many rows have been placed in each bin:

#count frequency of each bin

df['points\_bin'].value\_counts()

## Output/Results snippet:

(3.999, 10.667] 4

(10.667, 19.333] 4

(19.333, 25.0] 4

Name: points\_bin, dtype: int64

**References:**

1. <https://medium.com/@theclickreader/data-preprocessing-in-python-handling-missing-data-b717bcd4a264>
2. <https://www.geeksforgeeks.org/python-binning-method-for-data-smoothing/>
3. <https://www.statology.org/data-binning-in-python/>

# Activity

**Aim:** Finding outliers

**Learning outcome**:

**Duration**: 3 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

An Outlier is a data-item/object that deviates significantly from the rest of the (so-called normal)objects. They can be caused by measurement or execution errors. The analysis for outlier detection is referred to as outlier mining. There are many ways to detect the outliers, and the removal process is the data frame same as removing a data item from the panda’s data frame.

Here pandas data frame is used for a more realistic approach as in real-world project need to detect the outliers arouse during the data analysis step, the same approach can be used on lists and series-type objects.

Dataset:

Dataset used is Boston Housing dataset as it is preloaded in the sklearn library.

# Importing

import sklearn

from sklearn.datasets import load\_boston

import pandas as pd

import matplotlib.pyplot as plt

# Load the dataset

bos\_hou = load\_boston()

# Create the dataframe

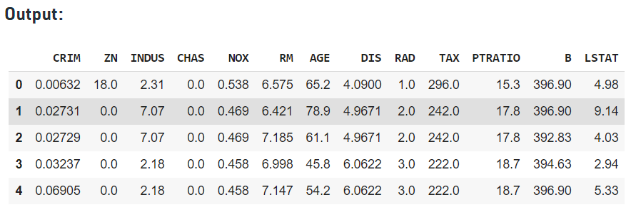
column\_name = bos\_hou.feature\_names

df\_boston = pd.DataFrame(bos\_hou.data)

df\_boston.columns = column\_name

df\_boston.head()

## Output/Results snippet:



**Detecting the outliers**

Outliers can be detected using visualization, implementing mathematical formulas on the dataset, or using the statistical approach. All of these are discussed below.

**1. Visualization**

**Example 1: Using Box Plot**

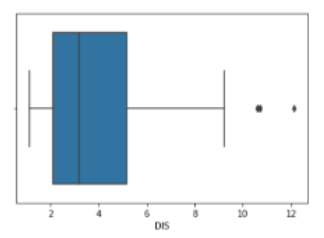
It captures the summary of the data effectively and efficiently with only a simple box and whiskers. Boxplot summarizes sample data using 25th, 50th, and 75th percentiles. One can just get insights(quartiles, median, and outliers) into the dataset by just looking at its boxplot.

# Box Plot

import seaborn as sns

sns.boxplot(df\_boston['DIS'])

## Output/Results snippet:



In the above graph, can clearly see that values above 10 are acting as the outliers.

# Position of the Outlier

print(np.where(df\_boston['DIS']>10))

## Output/Results snippet:



Example 2: Using ScatterPlot.

It is used when you have paired numerical data, or when your dependent variable has multiple values for each reading independent variable, or when trying to determine the relationship between the two variables. In the process of utilizing the scatter plot, one can also use it for outlier detection.

To plot the scatter plot one requires two variables that are somehow related to each other. So here, ‘Proportion of non-retail business acres per town’ and ‘Full-value property-tax rate per $10,000’ are used whose column names are “INDUS” and “TAX” respectively.

# Scatter plot

fig, ax = plt.subplots(figsize = (18,10))

ax.scatter(df\_boston['INDUS'], df\_boston['TAX'])

# x-axis label

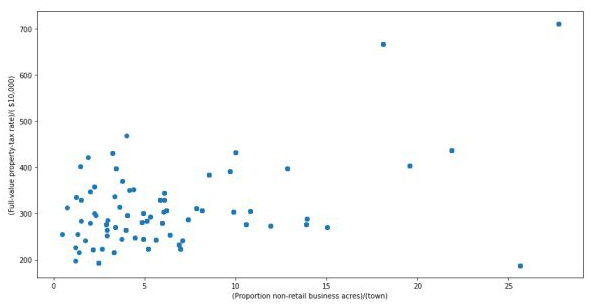
ax.set\_xlabel('(Proportion non-retail business acres)/(town)')

# y-axis label

ax.set\_ylabel('(Full-value property-tax rate)/( $10,000)')

plt.show()

## Output/Results snippet:



Looking at the graph can summarize that most of the data points are in the bottom left corner of the graph but there are few points that are exactly;y opposite that is the top right corner of the graph. Those points in the top right corner can be regarded as Outliers.

Using approximation can say all those data points that are x>20 and y>600 are outliers. The following code can fetch the exact position of all those points that satisfy these conditions.

# Position of the Outlier

print(np.where((df\_boston['INDUS']>20) & (df\_boston['TAX']>600)))

## Output/Results snippet:



**2. Z-score**

Z- Score is also called a standard score. This value/score helps to understand that how far is the data point from the mean. And after setting up a threshold value one can utilize z score values of data points to define the outliers.

Zscore = (data\_point -mean) / std. deviation

# Z score

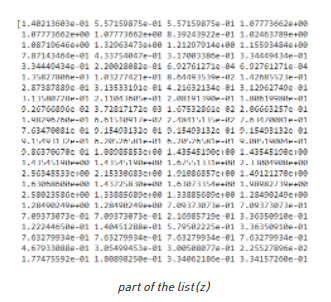
from scipy import stats

import numpy as np

z = np.abs(stats.zscore(df\_boston['DIS']))

print(z)

## Output/Results snippet:



The above output is just a snapshot of part of the data; the actual length of the list(z) is 506 that is the number of rows. It prints the z-score values of each data item of the column

Now to define an outlier threshold value is chosen which is generally 3.0. As 99.7% of the data points lie between +/- 3 standard deviation (using Gaussian Distribution approach).

threshold = 3

# Position of the outlier

print(np.where(z > 3))

## Output/Results snippet:



**3. IQR (Inter Quartile Range)**

IQR (Inter Quartile Range) Inter Quartile Range approach to finding the outliers is the most commonly used and most trusted approach used in the research field.

IQR = Quartile3 – Quartile1

# IQR

Q1 = np.percentile(df\_boston['DIS'], 25,

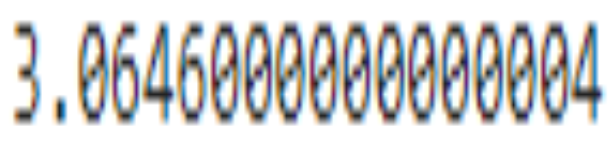
interpolation = 'midpoint')

Q3 = np.percentile(df\_boston['DIS'], 75,

interpolation = 'midpoint')

IQR = Q3 - Q1

## Output/Results snippet:



To define the outlier base value is defined above and below datasets normal range namely Upper and Lower bounds, define the upper and the lower bound (1.5\*IQR value is considered) :

upper = Q3 +1.5\*IQR

lower = Q1 – 1.5\*IQR

In the above formula as according to statistics, the 0.5 scale-up of IQR (new\_IQR = IQR + 0.5\*IQR) is taken, to consider all the data between 2.7 standard deviations in the Gaussian Distribution.

# Above Upper bound

upper = df\_boston['DIS'] >= (Q3+1.5\*IQR)

print("Upper bound:",upper)

print(np.where(upper))

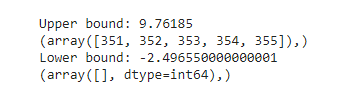
# Below Lower bound

lower = df\_boston['DIS'] <= (Q1-1.5\*IQR)

print("Lower bound:", lower)

print(np.where(lower))

## Output/Results snippet:



**Removing the outliers**

For removing the outlier, one must follow the same process of removing an entry from the dataset using its exact position in the dataset because in all the above methods of detecting the outliers end result is the list of all those data items that satisfy the outlier definition according to the method used.

dataframe.drop( row\_index, inplace = True

The above code can be used to drop a row from the dataset given the row\_indexes to be dropped. Inplace =True is used to tell python to make the required change in the original dataset. row\_index can be only one value or list of values or NumPy array but it must be one dimensional.

Example:

df\_boston.drop(lists[0],inplace = True)

Detecting the outliers using IQR and removing them.

# Importing

import sklearn

from sklearn.datasets import load\_boston

import pandas as pd

# Load the dataset

bos\_hou = load\_boston()

# Create the dataframe

column\_name = bos\_hou.feature\_names

df\_boston = pd.DataFrame(bos\_hou.data)

df\_boston.columns = column\_name

df\_boston.head()

''' Detection '''

# IQR

Q1 = np.percentile(df\_boston['DIS'], 25,

interpolation = 'midpoint')

Q3 = np.percentile(df\_boston['DIS'], 75,

interpolation = 'midpoint')

IQR = Q3 - Q1

print("Old Shape: ", df\_boston.shape)

# Upper bound

upper = np.where(df\_boston['DIS'] >= (Q3+1.5\*IQR))

# Lower bound

lower = np.where(df\_boston['DIS'] <= (Q1-1.5\*IQR))

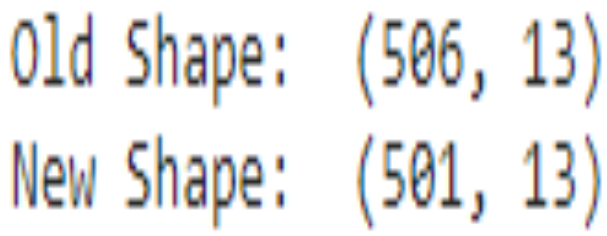
''' Removing the Outliers '''

df\_boston.drop(upper[0], inplace = True)

df\_boston.drop(lower[0], inplace = True)

print("New Shape: ", df\_boston.shape)

## Output/Results snippet:



**References:**

1. <https://www.geeksforgeeks.org/detect-and-remove-the-outliers-using-python/>

# Activity

**Aim:** Visualising your data through matplotlib under basic charts

**Learning outcome**:

**Duration**: 5 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

## Data Visualization is an important part of business activities as organizations nowadays collect a huge amount of data. Sensors all over the world are collecting climate data, user data through clicks, car data for prediction of steering wheels etc. All of these data collected hold key insights for businesses and visualizations make these insights easy to interpret.

## Matplotlib

## Matplotlib is a 2-D plotting library that helps in visualizing figures. Matplotlib emulates Matlab like graphs and visualizations. Matlab is not free, is difficult to scale and as a programming language is tedious. So, matplotlib in Python is used as it is a robust, free and easy library for data visualization.

## Anatomy of Matplotlib Figure

## 

## The figure contains the overall window where plotting happens, contained within the figure are where actual graphs are plotted. Every Axes has an x-axis and y-axis for plotting. And contained within the axes are titles, ticks, labels associated with each axis. An essential figure of matplotlib is that we can more than axes in a figure which helps in building multiple plots, as shown below. In matplotlib, pyplot is used to create figures and change the characteristics of figures.

## 

## Things to follow

## Plotting of Matplotlib is quite easy. Generally, while plotting they follow the same steps in each and every plot. Matplotlib has a module called pyplot which aids in plotting figure. The Jupyter notebook is used for running the plots. We import matplotlib.pyplot as plt for making it call the package module.

## Importing required libraries and dataset to plot using Pandas pd.read\_csv()

## Extracting important parts for plots using conditions on Pandas Dataframes.

## plt.plot()for plotting line chart similarly in place of plot other functions are used for plotting. All plotting functions require data and it is provided in the function through parameters.

## plot.xlabel , plt.ylabel for labeling x and y-axis respectively.

## plt.xticks , plt.yticks for labeling x and y-axis observation tick points respectively.

## plt.legend() for signifying the observation variables.

## plt.title() for setting the title of the plot.

## plot.show() for displaying the plot.

**Different types of Matplotlib Plots**

Matplotlib supports a variety of plots including line charts, bar charts, histograms, scatter plots, etc.

**Line Chart**

Line chart is one of the basic plots and can be created using the plot() function. It is used to represent a relationship between two data X and Y on a different axis.

Syntax:

matplotlib.pyplot.plot(\\*args, scalex=True, scaley=True, data=None, \\*\\*kwargs)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

plt.plot(x, y)

# Adding title to the plot

plt.title("Line Chart")

# Adding label on the y-axis

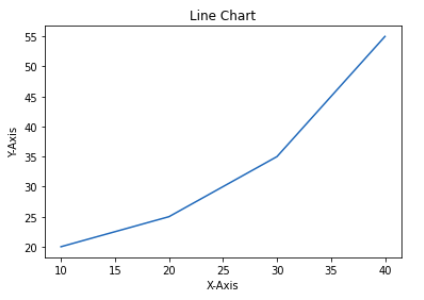
plt.ylabel('Y-Axis')

# Adding label on the x-axis

plt.xlabel('X-Axis')

plt.show()

## Output/Results snippet:



**Bar Chart**

A bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories. It can be created using the bar() method.

In the below example, we will use the tips dataset. Tips database is the record of the tip given by the customers in a restaurant for two and a half months in the early 1990s. It contains 6 columns as total\_bill, tip, sex, smoker, day, time, size.

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

x = data['day']

y = data['total\_bill']

# plotting the data

plt.bar(x, y)

# Adding title to the plot

plt.title("Tips Dataset")

# Adding label on the y-axis

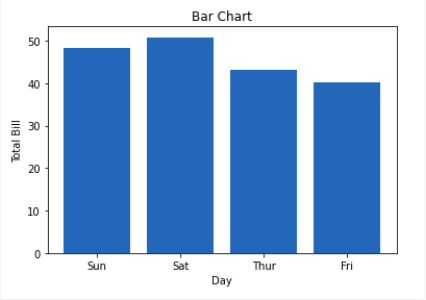
plt.ylabel('Total Bill')

# Adding label on the x-axis

plt.xlabel('Day')

plt.show()

## Output/Results snippet:



**Histogram**

A histogram is basically used to represent data provided in a form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. The hist() function is used to compute and create histogram of x.

Syntax:

matplotlib.pyplot.hist(x, bins=None, range=None, density=False, weights=None, cumulative=False, bottom=None, histtype=’bar’, align=’mid’, orientation=’vertical’, rwidth=None, log=False, color=None, label=None, stacked=False, \\*, data=None, \\*\\*kwargs)

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

x = data['total\_bill']

# plotting the data

plt.hist(x)

# Adding title to the plot

plt.title("Tips Dataset")

# Adding label on the y-axis

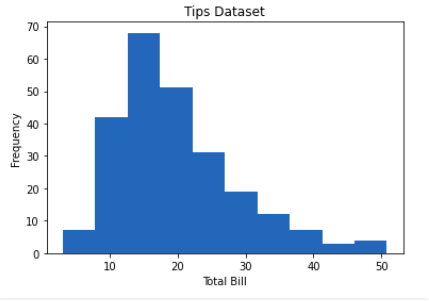
plt.ylabel('Frequency')

# Adding label on the x-axis

plt.xlabel('Total Bill')

plt.show()

## Output/Results snippet:



**Scatter Plot**

Scatter plots are used to observe relationships between variables. The scatter() method in the matplotlib library is used to draw a scatter plot.

Syntax:

matplotlib.pyplot.scatter(x\_axis\_data, y\_axis\_data, s=None, c=None, marker=None, cmap=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors=None

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

x = data['day']

y = data['total\_bill']

# plotting the data

plt.scatter(x, y)

# Adding title to the plot

plt.title("Tips Dataset")

# Adding label on the y-axis

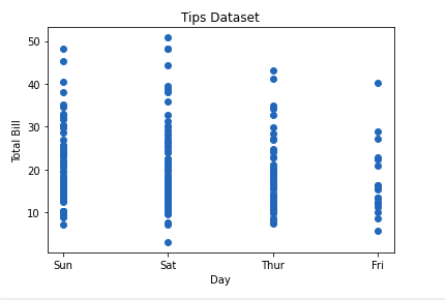
plt.ylabel('Total Bill')

# Adding label on the x-axis

plt.xlabel('Day')

plt.show()

## Output/Results snippet:



**Pie Chart**

Pie chart is a circular chart used to display only one series of data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges. It can be created using the pie() method.

Syntax:

matplotlib.pyplot.pie(data, explode=None, labels=None, colors=None, autopct=None, shadow=False)

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

cars = ['AUDI', 'BMW', 'FORD',

'TESLA', 'JAGUAR',]

data = [23, 10, 35, 15, 12]

# plotting the data

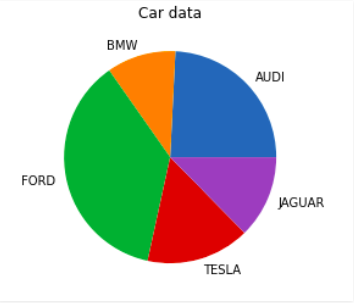
plt.pie(data, labels=cars)

# Adding title to the plot

plt.title("Car data")

plt.show()

## Output/Results snippet:



**Saving a Plot**

For saving a plot in a file on storage disk, savefig() method is used. A file can be saved in many formats like .png, .jpg, .pdf, etc.

Syntax:

pyplot.savefig(fname, dpi=None, facecolor=’w’, edgecolor=’w’, orientation=’portrait’, papertype=None, format=None, transparent=False, bbox\_inches=None, pad\_inches=0.1, frameon=None, metadata=None)

import matplotlib.pyplot as plt

# Creating data

year = ['2010', '2002', '2004', '2006', '2008']

production = [25, 15, 35, 30, 10]

# Plotting barchart

plt.bar(year, production)

# Saving the figure.

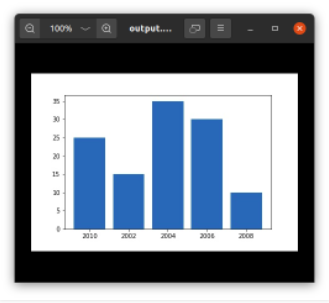
plt.savefig("output.jpg")

# Saving figure by changing parameter values

plt.savefig("output1", facecolor='y', bbox\_inches="tight",

pad\_inches=0.3, transparent=True)

## Output/Results snippet:





**References:**

1. <https://towardsdatascience.com/data-visualization-using-matplotlib-16f1aae5ce70>
2. <https://www.geeksforgeeks.org/data-visualization-using-matplotlib/>

# Activity

**Aim:** Labels, legends and axes

**Learning outcome**:

**Duration**: 3 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

**Adding Title**

The title() method in matplotlib module is used to specify the title of the visualization depicted and displays the title using various attributes.

Syntax:

matplotlib.pyplot.title(label, fontdict=None, loc=’center’, pad=None, \*\*kwargs)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

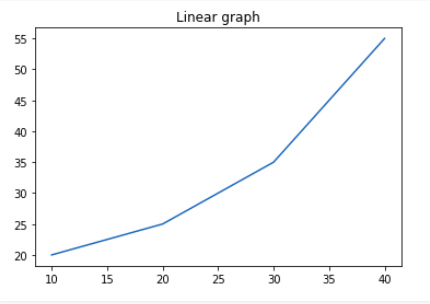
plt.plot(x, y)

# Adding title to the plot

plt.title("Linear graph")

plt.show()

## Output/Results snippet:



We can also change the appearance of the title by using the parameters of this function.

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

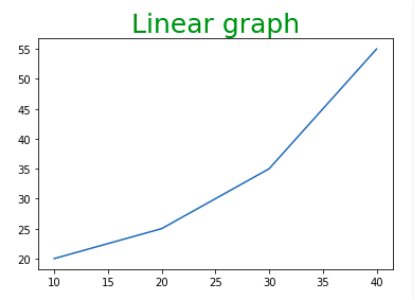
plt.plot(x, y)

# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")

plt.show()

## Output/Results snippet:



**Adding X Label and Y Label**

In layman’s terms, the X label and the Y label are the titles given to X-axis and Y-axis respectively. These can be added to the graph by using the xlabel() and ylabel() methods.

Syntax:

matplotlib.pyplot.xlabel(xlabel, fontdict=None, labelpad=None, \*\*kwargs)

matplotlib.pyplot.ylabel(ylabel, fontdict=None, labelpad=None, \*\*kwargs)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

plt.plot(x, y)

# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")

# Adding label on the y-axis

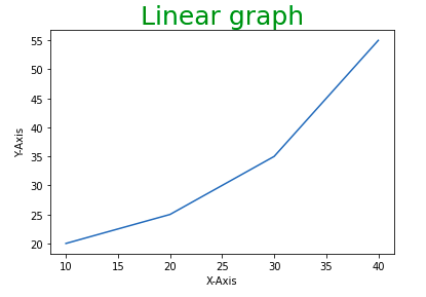
plt.ylabel('Y-Axis')

# Adding label on the x-axis

plt.xlabel('X-Axis')

plt.show()

## Output/Results snippet:



**Setting Limits and Tick labels**

You might have seen that Matplotlib automatically sets the values and the markers(points) of the X and Y axis, however, it is possible to set the limit and markers manually. xlim() and ylim() functions are used to set the limits of the X-axis and Y-axis respectively. Similarly, xticks() and yticks() functions are used to set tick labels.

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

plt.plot(x, y)

# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")

# Adding label on the y-axis

plt.ylabel('Y-Axis')

# Adding label on the x-axis

plt.xlabel('X-Axis')

# Setting the limit of y-axis

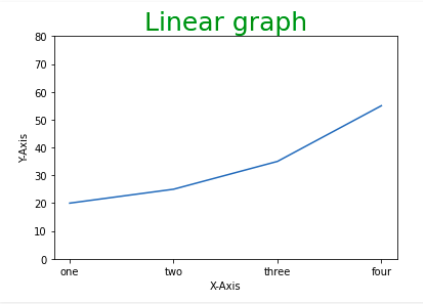
plt.ylim(0, 80)

# setting the labels of x-axis

plt.xticks(x, labels=["one", "two", "three", "four"])

plt.show()

## Output/Results snippet:



**Adding Legends**

A legend is an area describing the elements of the graph. In simple terms, it reflects the data displayed in the graph’s Y-axis. It generally appears as the box containing a small sample of each color on the graph and a small description of what this data means.

The attribute bbox\_to\_anchor=(x, y) of legend() function is used to specify the coordinates of the legend, and the attribute ncol represents the number of columns that the legend has. Its default value is 1.

Syntax:

matplotlib.pyplot.legend([“name1”, “name2”], bbox\_to\_anchor=(x, y), ncol=1)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

plt.plot(x, y)

# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="green")

# Adding label on the y-axis

plt.ylabel('Y-Axis')

# Adding label on the x-axis

plt.xlabel('X-Axis')

# Setting the limit of y-axis

plt.ylim(0, 80)

# setting the labels of x-axis

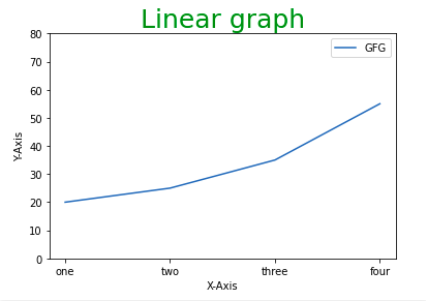
plt.xticks(x, labels=["one", "two", "three", "four"])

# Adding legends

plt.legend(["GFG"])

plt.show()

## Output/Results snippet:



**Figure class**

Consider the figure class as the overall window or page on which everything is drawn. It is a top-level container that contains one or more axes. A figure can be created using the figure() method.

Syntax:

class matplotlib.figure.Figure(figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=0.0, frameon=None, subplotpars=None, tight\_layout=None, constrained\_layout=None)

# Python program to show pyplot module

import matplotlib.pyplot as plt

from matplotlib.figure import Figure

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# Creating a new figure with width = 7 inches

# and height = 5 inches with face color as

# green, edgecolor as red and the line width

# of the edge as 7

fig = plt.figure(figsize =(7, 5), facecolor='g',

edgecolor='b', linewidth=7)

# Creating a new axes for the figure

ax = fig.add\_axes([1, 1, 1, 1])

# Adding the data to be plotted

ax.plot(x, y)

# Adding title to the plot

plt.title("Linear graph", fontsize=25, color="yellow")

# Adding label on the y-axis

plt.ylabel('Y-Axis')

# Adding label on the x-axis

plt.xlabel('X-Axis')

# Setting the limit of y-axis

plt.ylim(0, 80)

# setting the labels of x-axis

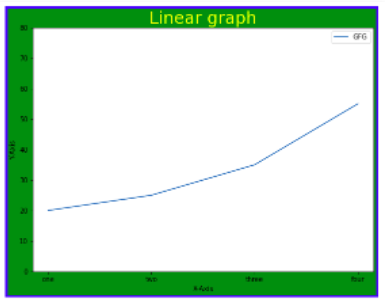
plt.xticks(x, labels=["one", "two", "three", "four"])

# Adding legends

plt.legend(["GFG"])

plt.show()

## Output/Results snippet:



**Axes Class**

Axes class is the most basic and flexible unit for creating sub-plots. A given figure may contain many axes, but a given axes can only be present in one figure. The axes() function creates the axes object.

Syntax:

axes([left, bottom, width, height])

Just like pyplot class, axes class also provides methods for adding titles, legends, limits, labels, etc.

* Adding Title – ax.set\_title()
* Adding X Label and Y label – ax.set\_xlabel(), ax.set\_ylabel()
* Setting Limits – ax.set\_xlim(), ax.set\_ylim()
* Tick labels – ax.set\_xticklabels(), ax.set\_yticklabels()
* Adding Legends – ax.legend()

# Python program to show pyplot module

import matplotlib.pyplot as plt

from matplotlib.figure import Figure

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

fig = plt.figure(figsize = (5, 4))

# Adding the axes to the figure

ax = fig.add\_axes([1, 1, 1, 1])

# plotting 1st dataset to the figure

ax1 = ax.plot(x, y)

# plotting 2nd dataset to the figure

ax2 = ax.plot(y, x)

# Setting Title

ax.set\_title("Linear Graph")

# Setting Label

ax.set\_xlabel("X-Axis")

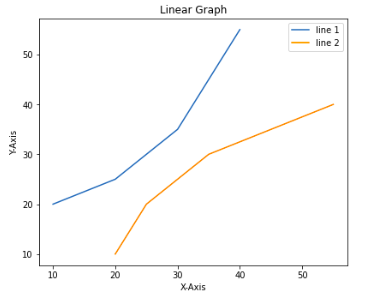
ax.set\_ylabel("Y-Axis")

# Adding Legend

ax.legend(labels = ('line 1', 'line 2'))

plt.show()

## Output/Results snippet:



**References:**

1. <https://www.geeksforgeeks.org/data-visualization-using-matplotlib/>

# Activity

**Aim:** Subplotting, grid, and 3D plots

**Learning outcome**:

**Duration**: 3 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

**Multiple Plots**

We have learned about the basic components of a graph that can be added so that it can convey more information. One method can be by calling the plot function again and again with a different set of values as shown in the above example. Now let’s see how to plot multiple graphs using some functions and also how to plot subplots.

**Method 1: Using the add\_axes() method**

The add\_axes() method is used to add axes to the figure. This is a method of figure class

Syntax:

add\_axes(self, \*args, \*\*kwargs)

# Python program to show pyplot module

import matplotlib.pyplot as plt

from matplotlib.figure import Figure

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# Creating a new figure with width = 5 inches

# and height = 4 inches

fig = plt.figure(figsize =(5, 4))

# Creating first axes for the figure

ax1 = fig.add\_axes([0.1, 0.1, 0.8, 0.8])

# Creating second axes for the figure

ax2 = fig.add\_axes([1, 0.1, 0.8, 0.8])

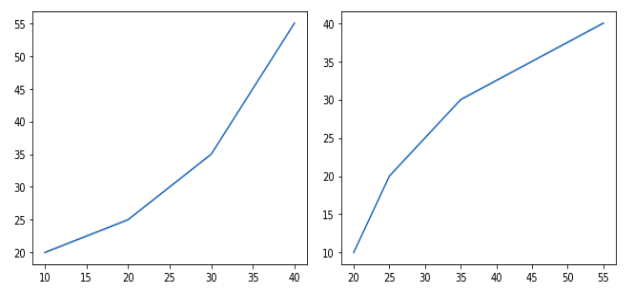
# Adding the data to be plotted

ax1.plot(x, y)

ax2.plot(y, x)

plt.show()

## Output/Results snippet:



**Method 2: Using subplot() method.**

This method adds another plot at the specified grid position in the current figure.

Syntax:

subplot(nrows, ncols, index, \*\*kwargs)

subplot(pos, \*\*kwargs)

subplot(ax)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# Creating figure object

plt.figure()

# addind first subplot

plt.subplot(121)

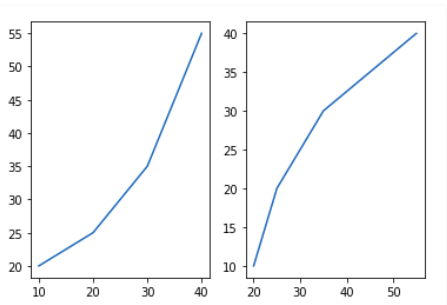
plt.plot(x, y)

# addding second subplot

plt.subplot(122)

plt.plot(y, x)

## Output/Results snippet:



**Method 3: Using subplots() method**

This function is used to create figures and multiple subplots at the same time.

Syntax:

matplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot\_kw=None, gridspec\_kw=None, \*\*fig\_kw)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# Creating the figure and subplots

# according the argument passed

fig, axes = plt.subplots(1, 2)

# plotting the data in the

# 1st subplot

axes[0].plot(x, y)

# plotting the data in the 1st

# subplot only

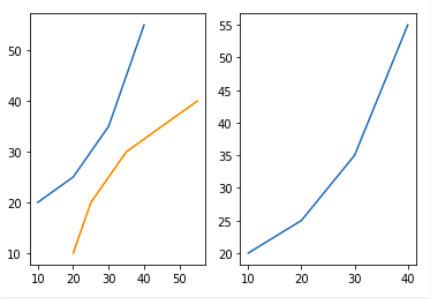
axes[0].plot(y, x)

# plotting the data in the 2nd

# subplot only

axes[1].plot(x, y)

## Output/Results snippet:



**Method 4: Using subplot2grid() method**

This function creates axes object at a specified location inside a grid and also helps in spanning the axes object across multiple rows or columns. In simpler words, this function is used to create multiple charts within the same figure.

Syntax:

Plt.subplot2grid(shape, location, rowspan, colspan)

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# adding the subplots

axes1 = plt.subplot2grid (

(7, 1), (0, 0), rowspan = 2, colspan = 1)

axes2 = plt.subplot2grid (

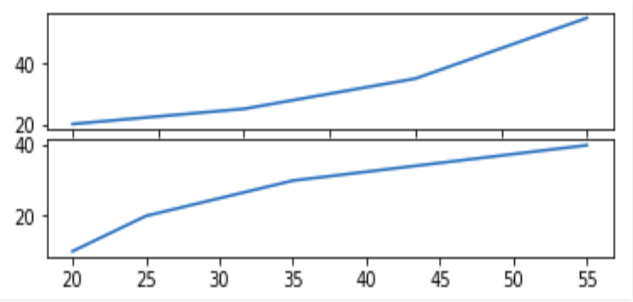
(7, 1), (2, 0), rowspan = 2, colspan = 1)

# plotting the data

axes1.plot(x, y)

axes2.plot(y, x)

## Output/Results snippet:



**Grids in Matplotlib**

Grids are made up of intersecting straight (vertical, horizontal and angular) or curved lines used to structure our content. Matplotlib helps us to draw plain graphs but it sometimes necessary to use grids for better understanding and get a reference for our data points. Thus, Matplotlib provides a grid() for easy creation of gridlines with tonnes of customization.

matplotlib.pyplot.grid()

Syntax: matplotlib.pyplot.grid(b=None, which=’major’, axis=’both’, \*\*kwargs)

## Program:

Parameters:

b: bool value to specify whether to show grid-lines. Default is True

which: The grid lines to apply changes. Values: {‘major’, ‘minor’, ‘both’}

axis: The axis to apply changes on. Values: {‘both’, ‘x’, ‘y’}

\*\*kwargs: Optional line properties

Returns: This function doesn’t return anything.

The grid() sets the visibility of grids by specifying a boolean value (True/False). We can also choose to display minor or major ticks or both. Also, color, linewidth and linestyle can be changed as additional parameters.

# Implementation of matplotlib function

import matplotlib.pyplot as plt

import numpy as np

# dummy data

x1 = np.linspace(0.0, 5.0)

y1 = np.cos(2 \* np.pi \* x1) \* np.exp(-x1)

# creates two subplots

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (12, 5))

# Plot without grid

ax1.plot(x1, y1)

ax1.set\_title('Plot without grid')

# plot with grid

ax2.plot(x1, y1)

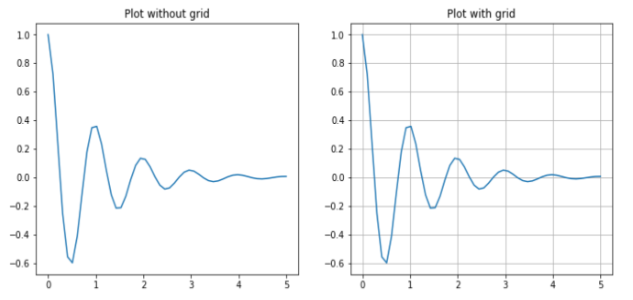
ax2.set\_title("Plot with grid")

# draw gridlines

ax2.grid(True)

plt.show()

## Output/Results snippet:



Now let’s draw gridlines using extra line properties such as color, linestyle and linewidth.

## Program:

# Implementation of matplotlib function

import matplotlib.pyplot as plt

import numpy as np

# dummy data

x = np.linspace(0, 2 \* np.pi, 400)

y = np.sin(x \*\* 2)

# set graph color

plt.plot(x, y, 'green')

# to set title

plt.title("Plot with linewidth and linestyle")

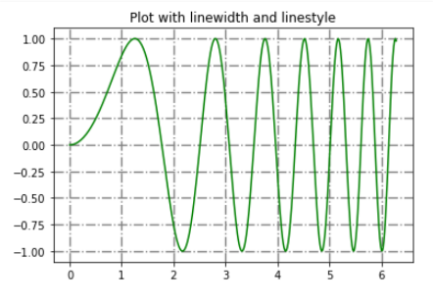
# draws gridlines of grey color using given

# linewidth and linestyle

plt.grid(True, color = "grey", linewidth = "1.4", linestyle = "-.")

plt.show()

## Output/Results snippet:



**3D – Plotting**

## Program:

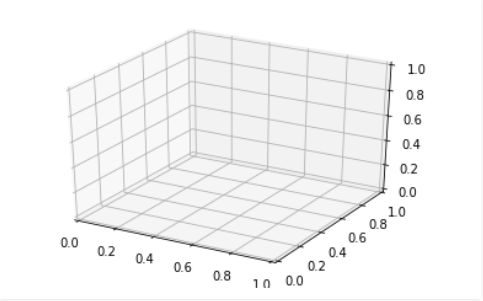
## import numpy as np

## import matplotlib.pyplot as plt

## fig = plt.figure()

## ax = plt.axes(projection ='3d')

## Output/Results snippet:



**Plotting 3-D Lines and Points**

Graph with lines and point are the simplest 3 dimensional graph. ax.plot3d and ax.scatter are the function to plot line and point graph respectively.

**3 dimensional line graph**

## Program:

# importing mplot3d toolkits, numpy and matplotlib

from mpl\_toolkits import mplot3d

import numpy as np

import matplotlib.pyplot as plt

fig = plt.figure()

# syntax for 3-D projection

ax = plt.axes(projection ='3d')

# defining all 3 axes

z = np.linspace(0, 1, 100)

x = z \* np.sin(25 \* z)

y = z \* np.cos(25 \* z)

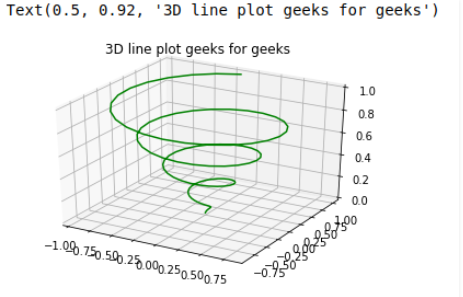
# plotting

ax.plot3D(x, y, z, 'green')

ax.set\_title('3D line plot geeks for geeks')

plt.show()

## Output/Results snippet:



## 3 dimensional scattered graph

## Program:

# importing mplot3d toolkits

from mpl\_toolkits import mplot3d

import numpy as np

import matplotlib.pyplot as plt

fig = plt.figure()

# syntax for 3-D projection

ax = plt.axes(projection ='3d')

# defining axes

z = np.linspace(0, 1, 100)

x = z \* np.sin(25 \* z)

y = z \* np.cos(25 \* z)

c = x + y

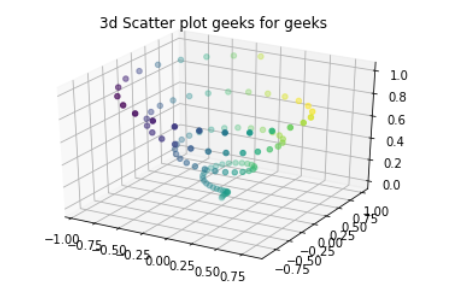
ax.scatter(x, y, z, c = c)

# syntax for plotting

ax.set\_title('3d Scatter plot geeks for geeks')

plt.show()

## Output/Results snippet:



**References:**

1. <https://www.geeksforgeeks.org/data-visualization-using-matplotlib/>
2. <https://www.geeksforgeeks.org/grids-in-matplotlib/>
3. <https://www.geeksforgeeks.org/three-dimensional-plotting-in-python-using-matplotlib/>

# Activity

**Aim:** Plot formatting- custom attribute values

**Learning outcome**:

**Duration**: 4 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

**Line Chart**

We may use the following properties –

* color: Changing the color of the line
* linewidth: Cutomizing the width of the line
* marker: For changing the style of actual plotted point
* markersize: For changing the size of the markers
* linestyle: For defining the style of the plotted line

**Different Linestyle available**

| **Character** | **Definition** |
| --- | --- |
| **–** | Solid line |
| **—** | Dashed line |
| **-.** | dash-dot line |
| **:** | Dotted line |
| **.** | Point marker |
| **o** | Circle marker |
| **,** | Pixel marker |
| **v** | triangle\_down marker |
| **^** | triangle\_up marker |
| **<** | triangle\_left marker |
| **>** | triangle\_right marker |
| **1** | tri\_down marker |
| **2** | tri\_up marker |
| **3** | tri\_left marker |
| **4** | tri\_right marker |
| **s** | square marker |
| **p** | pentagon marker |
| **\*** | star marker |
| **h** | hexagon1 marker |
| **H** | hexagon2 marker |
| **+** | Plus marker |
| **x** | X marker |
| **D** | Diamond marker |
| **d** | thin\_diamond marker |
| **|** | vline marker |
| **\_** | hline marker |

import matplotlib.pyplot as plt

# initializing the data

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

# plotting the data

plt.plot(x, y, color='green', linewidth=3, marker='o',

markersize=15, linestyle='--')

# Adding title to the plot

plt.title("Line Chart")

# Adding label on the y-axis

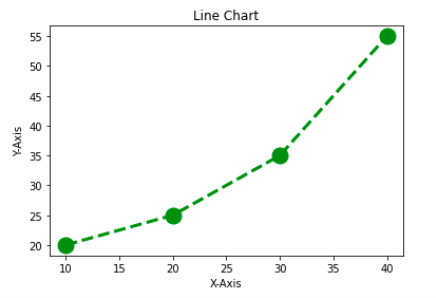
plt.ylabel('Y-Axis')

# Adding label on the x-axis

plt.xlabel('X-Axis')

plt.show()

**Output/Results snippet:**



**Bar Chart**

Customization that is available for the Bar Chart –

* color: For the bar faces
* edgecolor: Color of edges of the bar
* linewidth: Width of the bar edges
* width: Width of the bar

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

x = data['day']

y = data['total\_bill']

# plotting the data

plt.bar(x, y, color='green', edgecolor='blue',

linewidth=2)

# Adding title to the plot

plt.title("Tips Dataset")

# Adding label on the y-axis

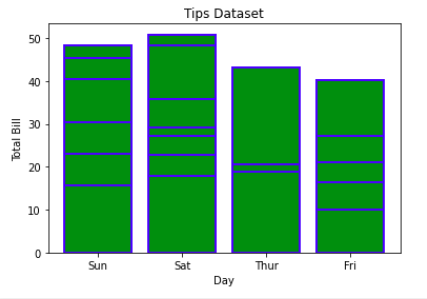
plt.ylabel('Total Bill')

# Adding label on the x-axis

plt.xlabel('Day')

plt.show()

**Output/Results snippet:**



Note: The lines in between the bars refer to the different values in the Y-axis of the particular value of the X-axis.

**Histogram**

Customization that is available for the Histogram –

* bins: Number of equal-width bins
* color: For changing the face color
* edgecolor: Color of the edges
* linestyle: For the edgelines
* alpha: blending value, between 0 (transparent) and 1 (opaque)

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

x = data['total\_bill']

# plotting the data

plt.hist(x, bins=25, color='green', edgecolor='blue',

linestyle='--', alpha=0.5)

# Adding title to the plot

plt.title("Tips Dataset")

# Adding label on the y-axis

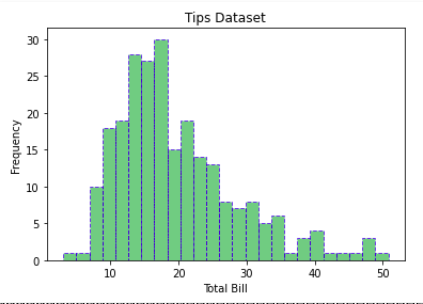
plt.ylabel('Frequency')

# Adding label on the x-axis

plt.xlabel('Total Bill')

plt.show()

**Output/Results snippet:**



**Scatter Plot**

Customizations that are available for the scatter plot are –

* s: marker size (can be scalar or array of size equal to size of x or y)
* c: color of sequence of colors for markers
* marker: marker style
* linewidths: width of marker border
* edgecolor: marker border color
* alpha: blending value, between 0 (transparent) and 1 (opaque)

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

x = data['day']

y = data['total\_bill']

# plotting the data

plt.scatter(x, y, c=data['size'], s=data['total\_bill'],

marker='D', alpha=0.5)

# Adding title to the plot

plt.title("Tips Dataset")

# Adding label on the y-axis

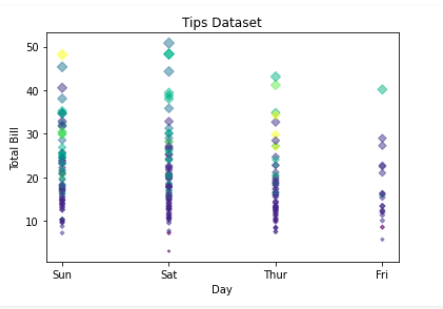
plt.ylabel('Total Bill')

# Adding label on the x-axis

plt.xlabel('Day')

plt.show()

**Output/Results snippet:**



**Pie Chart**

Customizations that are available for the Pie chart are –

* explode: Moving the wedges of the plot
* autopct: Label the wedge with their numerical value.
* color: Attribute is used to provide color to the wedges.
* shadow: Used to create shadow of wedge.

import matplotlib.pyplot as plt

import pandas as pd

# Reading the tips.csv file

data = pd.read\_csv('tips.csv')

# initializing the data

cars = ['AUDI', 'BMW', 'FORD',

'TESLA', 'JAGUAR',]

data = [23, 13, 35, 15, 12]

explode = [0.1, 0.5, 0, 0, 0]

colors = ( "orange", "cyan", "yellow",

"grey", "green",)

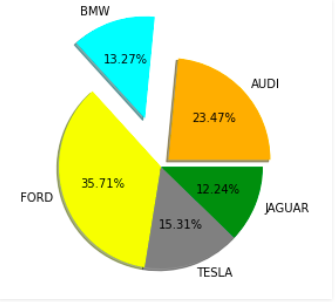
# plotting the data

plt.pie(data, labels=cars, explode=explode, autopct='%1.2f%%',

colors=colors, shadow=True)

plt.show()

**Output/Results snippet:**



**References:**

1. <https://www.geeksforgeeks.org/data-visualization-using-matplotlib/>

# Activity

**Aim:** Advanced charts in seaborn- countplot(), jointplot(), boxplot(), heatmap(), regression plot, etc

**Learning outcome**:

**Duration**: 6 hours

## List of Hardware/Software requirements:

1. Laptop/Computer with Windows OS / Ubuntu
2. Python / Jupyter notebook

## Program:

Seaborn is a library mostly used for statistical plotting in Python. It is built on top of Matplotlib and provides beautiful default styles and color palettes to make statistical plots more attractive.

Seaborn can be installed using the pip. Type the below command in the terminal.

pip install seaborn

Plotting categorical scatter plots with Seaborn

**Stripplot**

# Python program to illustrate

# Plotting categorical scatter

# plots with Seaborn

# importing the required module

import matplotlib.pyplot as plt

import seaborn as sns

# x axis values

x =['sun', 'mon', 'fri', 'sat', 'tue', 'wed', 'thu']

# y axis values

y =[5, 6.7, 4, 6, 2, 4.9, 1.8]

# plotting strip plot with seaborn

ax = sns.stripplot(x, y);

# giving labels to x-axis and y-axis

ax.set(xlabel ='Days', ylabel ='Amount\_spend')

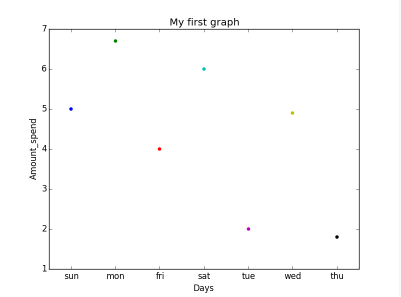
# giving title to the plot

plt.title('My first graph');

# function to show plot

plt.show()

## Output/Results snippet:



This is the one kind of scatter plot of categorical data with the help of seaborn.

* Categorical data is represented on the x-axis and values correspond to them represented through the y-axis.
* .striplot() function is used to define the type of the plot and to plot them on canvas using.
* .set() function is used to set labels of x-axis and y-axis.
* .title() function is used to give a title to the graph.
* To view plot we use .show() function.

Stripplot using inbuilt data-set given in seaborn :

# Python program to illustrate

# Stripplot using inbuilt data-set

# given in seaborn

# importing the required module

import matplotlib.pyplot as plt

import seaborn as sns

# use to set style of background of plot

sns.set(style="whitegrid")

# loading data-set

iris = sns.load\_dataset('iris')

# plotting strip plot with seaborn

# deciding the attributes of dataset on

# which plot should be made

ax = sns.stripplot(x='species', y='sepal\_length', data=iris)

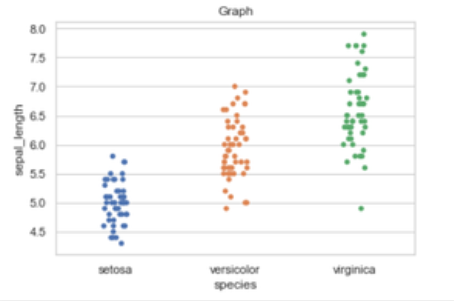
# giving title to the plot

plt.title('Graph')

# function to show plot

plt.show()

## Output/Results snippet:



Explanation:

* iris is the dataset already present in seaborn module for use.
* We use .load\_dataset() function in order to load the data.We can also load any other file by giving the path and name of the file in the argument.
* .set(style=”whitegrid”) function here is also use to define the background of plot.We can use “darkgrid”
* instead of whitegrid if we want the dark-colored background.
* In .stripplot() function we have to define which attribute of the dataset to be on the x-axis and which attribute of the dataset should on y-axis.data = iris means attributes which we define earlier should be taken from the given data.
* We can also draw this plot with matplotlib but the problem with matplotlib is its default parameters. The reason why Seaborn is so great with DataFrames is, for example, labels from DataFrames are automatically propagated to plots or other data structures as you see in the above figure column name species comes on the x-axis and column name stepal\_length comes on the y-axis, that is not possible with matplotlib. We have to explicitly define the labels of the x-axis and y-axis.

**Swarmplot**

# Python program to illustrate

# plotting using Swarmplot

# importing the required module

import matplotlib.pyplot as plt

import seaborn as sns

# use to set style of background of plot

sns.set(style="whitegrid")

# loading data-set

iris = sns.load\_dataset('iris')

# plotting strip plot with seaborn

# deciding the attributes of dataset on

# which plot should be made

ax = sns.swarmplot(x='species', y='sepal\_length', data=iris)

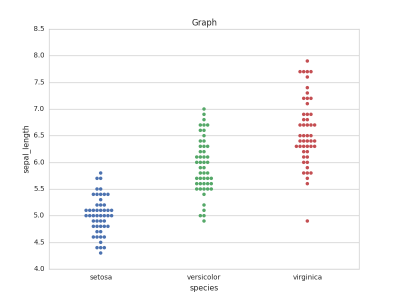
# giving title to the plot

plt.title('Graph')

# function to show plot

plt.show()

## Output/Results snippet:



Explanation:

This is very much similar to stripplot but the only difference is that it does not allow overlapping of markers. It causes jittering in the markers of the plot so that graph can easily be read without information loss as seen in the above plot.

* We use .swarmplot() function to plot swarn plot.
* Another difference that we can notice in Seaborn and Matplotlib is that working with DataFrames doesn’t go quite as smoothly with Matplotlib, which can be annoying if we doing exploratory analysis with Pandas. And that’s exactly what Seaborn does easily, the plotting functions operate on DataFrames and arrays that contain a whole dataset.

# importing the required module

import matplotlib.pyplot as plt

import seaborn as sns

# use to set style of background of plot

sns.set(style="whitegrid")

# loading data-set

iris = sns.load\_dataset('iris')

# plotting strip plot with seaborn

# deciding the attributes of dataset on

# which plot should be made

ax = sns.swarmplot(x='sepal\_length', y='species', data=iris)

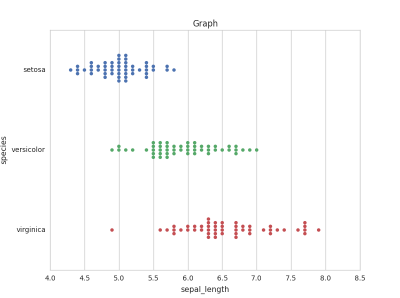
# giving title to the plot

plt.title('Graph')

# function to show plot

plt.show()

## Output/Results snippet:



**Barplot**

A barplot is basically used to aggregate the categorical data according to some methods and by default it’s the mean. It can also be understood as a visualization of the group by action. To use this plot we choose a categorical column for the x-axis and a numerical column for the y-axis, and we see that it creates a plot taking a mean per categorical column.

Syntax:

barplot([x, y, hue, data, order, hue\_order, …])

# import the seaborn library

import seaborn as sns

# reading the dataset

df = sns.load\_dataset('tips')

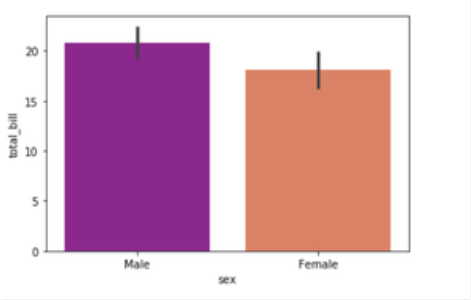
# change the estimator from mean to

# standard deviation

sns.barplot(x ='sex', y ='total\_bill', data = df,

palette ='plasma')

## Output/Results snippet:



Explanation:

Looking at the plot we can say that the average total\_bill for the male is more than compared to the female.

* Palette is used to set the color of the plot
* The estimator is used as a statistical function for estimation within each categorical bin.

**Countplot**

A countplot basically counts the categories and returns a count of their occurrences. It is one of the simplest plots provided by the seaborn library.

Syntax:

countplot([x, y, hue, data, order, …])

# import the seaborn library

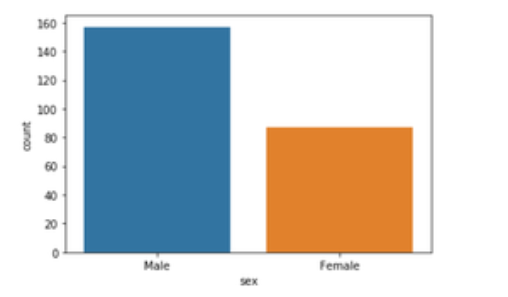
import seaborn as sns

# reading the dataset

df = sns.load\_dataset('tips')

sns.countplot(x ='sex', data = df)

## Output/Results snippet:



Explanation:

Looking at the plot we can say that the number of males is more than the number of females in the dataset. As it only returns the count based on a categorical column, we need to specify only the x parameter.

Boxplot

Box Plot is the visual representation of the depicting groups of numerical data through their quartiles. Boxplot is also used to detect the outlier in the data set.

Syntax:

boxplot([x, y, hue, data, order, hue\_order, …])

# import the seaborn library

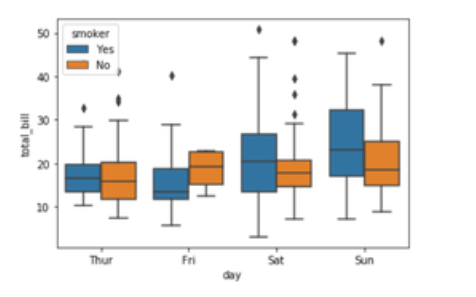
import seaborn as sns

# reading the dataset

df = sns.load\_dataset('tips')

sns.boxplot(x='day', y='total\_bill', data=df, hue='smoker')

## Output/Results snippet:



Explanation:

x takes the categorical column and y is a numerical column. Hence we can see the total bill spent each day.” hue” parameter is used to further add a categorical separation. By looking at the plot we can say that the people who do not smoke had a higher bill on Friday as compared to the people who smoked.

**Violinplot**

It is similar to the boxplot except that it provides a higher, more advanced visualization and uses the kernel density estimation to give a better description about the data distribution.

Syntax:

violinplot([x, y, hue, data, order, …])

# import the seaborn library

import seaborn as sns

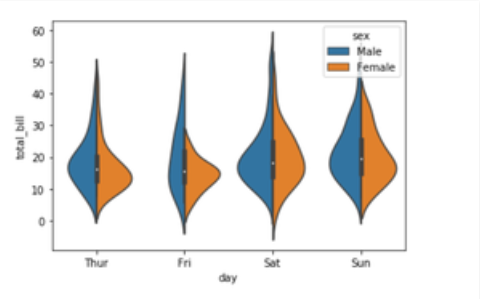
# reading the dataset

df = sns.load\_dataset('tips')

sns.violinplot(x='day', y='total\_bill', data=df,

hue='sex', split=True)

## Output/Results snippet:



Explanation:

* hue is used to separate the data further using the sex category
* setting split=True will draw half of a violin for each level. This can make it easier to directly compare the distributions.

**Stripplot**

It basically creates a scatter plot based on the category.

Syntax:

stripplot([x, y, hue, data, order, …])

# import the seaborn library

import seaborn as sns

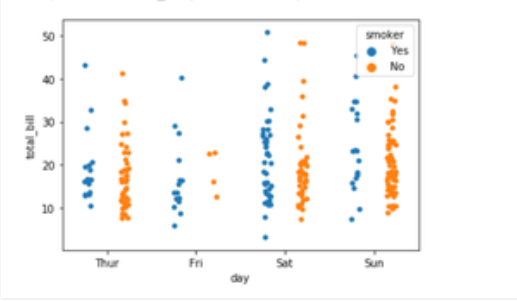
# reading the dataset

df = sns.load\_dataset('tips')

sns.stripplot(x='day', y='total\_bill', data=df,

jitter=True, hue='smoker', dodge=True)

## Output/Results snippet:



Explanation:

* One problem with strip plot is that you can’t really tell which points are stacked on top of each other and hence we use the jitter parameter to add some random noise.
* jitter parameter is used to add an amount of jitter (only along the categorical axis) which can be useful when you have many points and they overlap so that it is easier to see the distribution.
* hue is used to provide an additional categorical separation
* setting split=True is used to draw separate strip plots based on the category specified by the hue parameter.

**Heatmap**

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. In this, to represent more common values or higher activities brighter colors basically reddish colors are used and to represent less common or activity values, darker colors are preferred. Heatmap is also defined by the name of the shading matrix. Heatmaps in Seaborn can be plotted by using the seaborn.heatmap() function.

seaborn.heatmap()

Syntax: seaborn.heatmap(data, \*, vmin=None, vmax=None, cmap=None, center=None, annot\_kws=None, linewidths=0, linecolor=’white’, cbar=True, \*\*kwargs)

Important Parameters:

* data: 2D dataset that can be coerced into an ndarray.
* vmin, vmax: Values to anchor the colormap, otherwise they are inferred from the data and other keyword arguments.
* cmap: The mapping from data values to color space.
* center: The value at which to center the colormap when plotting divergent data.
* annot: If True, write the data value in each cell.
* fmt: String formatting code to use when adding annotations.
* linewidths: Width of the lines that will divide each cell.
* linecolor: Color of the lines that will divide each cell.
* cbar: Whether to draw a colorbar.

All the parameters except data are optional.

Returns: An object of type matplotlib.axes.\_subplots.AxesSubplot

**Basic Heatmap**

Making a heatmap with the default parameters. We will be creating a 10×10 2-D data using the randint() function of the NumPy module.

# importing the modules

import numpy as np

import seaborn as sn

import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers

# from 1 to 100

data = np.random.randint(low = 1,

high = 100,

size = (10, 10))

print("The data to be plotted:\n")

print(data)

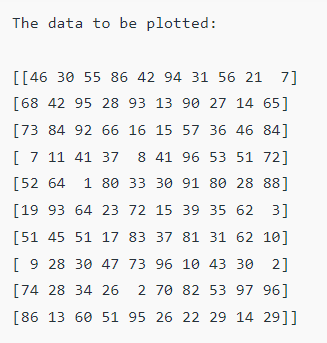
# plotting the heatmap

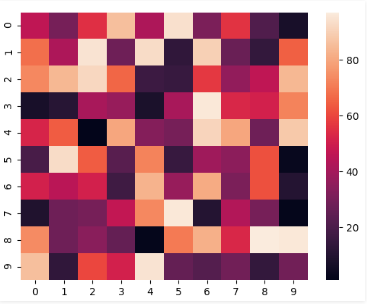
hm = sn.heatmap(data = data)

# displaying the plotted heatmap

plt.show()

## Output/Results snippet:





**Anchoring the colormap**

If we set the vmin value to 30 and the vmax value to 70, then only the cells with values between 30 and 70 will be displayed. This is called anchoring the colormap.

# importing the modules

import numpy as np

import seaborn as sn

import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers

# from 1 to 100

data = np.random.randint(low=1,

high=100,

size=(10, 10))

# setting the parameter values

vmin = 30

vmax = 70

# plotting the heatmap

hm = sn.heatmap(data=data,

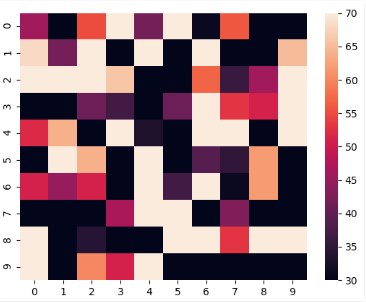
vmin=vmin,

vmax=vmax)

# displaying the plotted heatmap

plt.show()

## Output/Results snippet:



**Choosing the colormap**

In this, we will be looking at the cmap parameter. Matplotlib provides us with multiple colormaps, you can look at all of them here. In our example, we’ll be using tab20.

# importing the modules

import numpy as np

import seaborn as sn

import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers

# from 1 to 100

data = np.random.randint(low=1,

high=100,

size=(10, 10))

# setting the parameter values

cmap = "tab20"

# plotting the heatmap

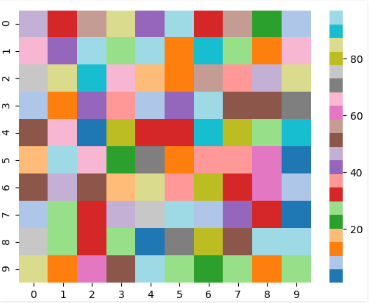
hm = sn.heatmap(data=data,

cmap=cmap)

# displaying the plotted heatmap

plt.show()

## Output/Results snippet:



**Displaying the cell values**

If we want to display the value of the cells, then we pass the parameter annot as True. fmt is used to select the datatype of the contents of the cells displayed.

# importing the modules

import numpy as np

import seaborn as sn

import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers

# from 1 to 100

data = np.random.randint(low=1,

high=100,

size=(10, 10))

# setting the parameter values

annot = True

# plotting the heatmap

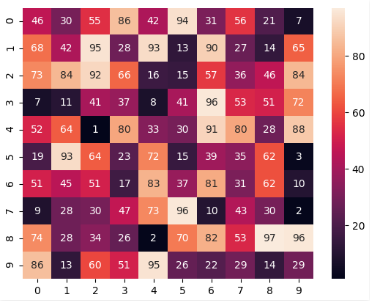
hm = sn.heatmap(data=data,

annot=annot)

# displaying the plotted heatmap

plt.show()

## Output/Results snippet:



**Customizing the separating line**

We can change the thickness and the color of the lines separating the cells using the linewidths and linecolor parameters respectively.

# importing the modules

import numpy as np

import seaborn as sn

import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers

# from 1 to 100

data = np.random.randint(low=1,

high=100,

size=(10, 10))

# setting the parameter values

linewidths = 2

linecolor = "yellow"

# plotting the heatmap

hm = sn.heatmap(data=data,

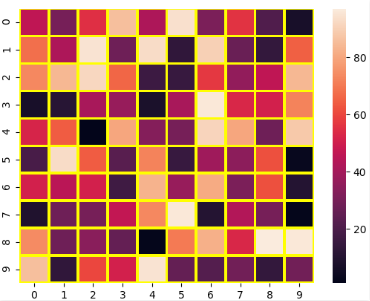
linewidths=linewidths,

linecolor=linecolor)

# displaying the plotted heatmap

plt.show()

## Output/Results snippet:



**Regression** **Plot**

The regression plots in seaborn are primarily intended to add a visual guide that helps to emphasize patterns in a dataset during exploratory data analyses. Regression plots as the name suggests creates a regression line between 2 parameters and helps to visualize their linear relationships.

Load the dataset

# import the library

import seaborn as sns

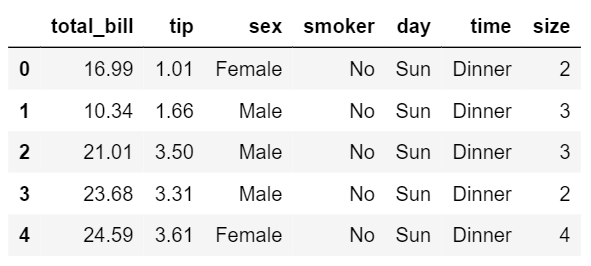
# load the dataset

dataset = sns.load\_dataset('tips')

# the first five entries of the dataset

dataset.head()

## Output/Results snippet:



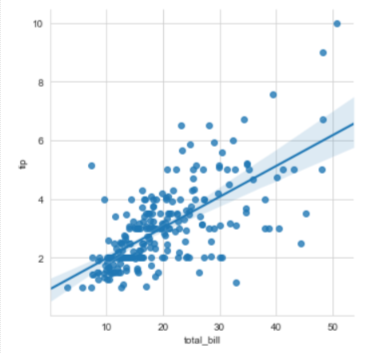
Regression plots in seaborn can be easily implemented with the help of the lmplot() function. lmplot() can be understood as a function that basically creates a linear model plot. lmplot() makes a very simple linear regression plot.It creates a scatter plot with a linear fit on top of it.

**Simple linear plot**

sns.set\_style('whitegrid')

sns.lmplot(x ='total\_bill', y ='tip', data = dataset)

## Output/Results snippet:



Explanation

x and y parameters are specified to provide values for the x and y axes. sns.set\_style() is used to have a grid in the background instead of a default white background. The data parameter is used to specify the source of information for drawing the plots.

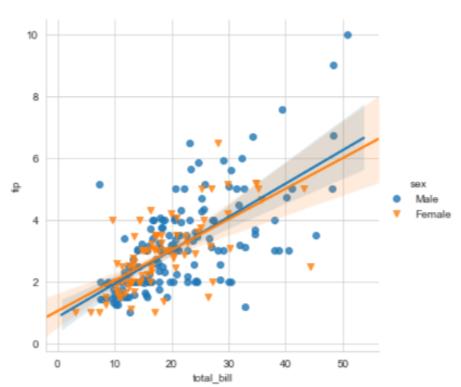
**Linear plot with additional parameters**

sns.set\_style('whitegrid')

sns.lmplot(x ='total\_bill', y ='tip', data = dataset,

hue ='sex', markers =['o', 'v'])

## Output/Results snippet:



Explanation

In order to have a better analysis capability using these plots, we can specify hue to have a categorical separation in our plot as well as use markers that come from the matplotlib marker symbols. Since we have two separate categories we need to pass in a list of symbols while specifying the marker.

**Setting the size and color of the plot**

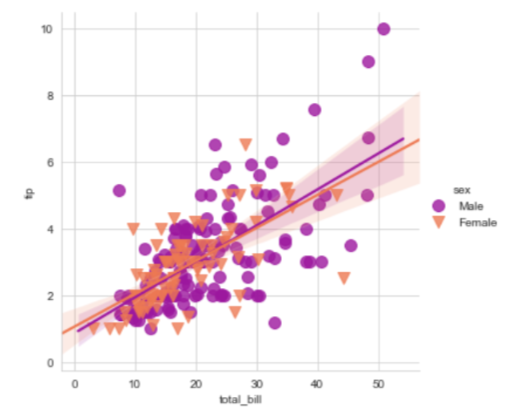
sns.set\_style('whitegrid')

sns.lmplot(x ='total\_bill', y ='tip', data = dataset, hue ='sex',

markers =['o', 'v'], scatter\_kws ={'s':100},

palette ='plasma')

## Output/Results snippet:



Explanation

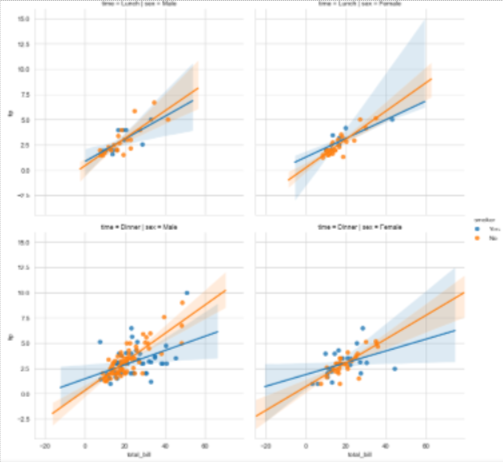
In this example what seaborn is doing is that its calling the matplotlib parameters indirectly to affect the scatter plots. We specify a parameter called scatter\_kws. We must note that the scatter\_kws parameter changes the size of only the scatter plots and not the regression lines. The regression lines remain untouched. We also use the palette parameter to change the color of the plot.

**Displaying multiple plots**

sns.lmplot(x ='total\_bill', y ='tip', data = dataset,

col ='sex', row ='time', hue ='smoker')

## Output/Results snippet:



**References:**

1. <https://www.geeksforgeeks.org/plotting-graph-using-seaborn-python/?ref=lbp>
2. <https://www.geeksforgeeks.org/seaborn-regression-plots/>