#Experiment No : 03

##**Aim :** Learn basics of maplotlib library and its use as visualisation tool in data science pipeline and learn about line plots and scatter plots.

Theory :

Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.

It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish.

This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large user base, which in turn has led to an active developer base and Matplotlib's powerful tools and ubiquity within the scientific Python world.

In recent years, however, the interface and style of Matplotlib have begun to show their age. Newer tools like ggplot and ggvis in the R language, along with web visualization toolkits based on D3js and HTML5 canvas, often make Matplotlib feel clunky and old-fashioned.

Performance :

[Students need to execute eaxh and every cell in this section and note the output of the same. Once done they have to answer Questions mentioned in review section]

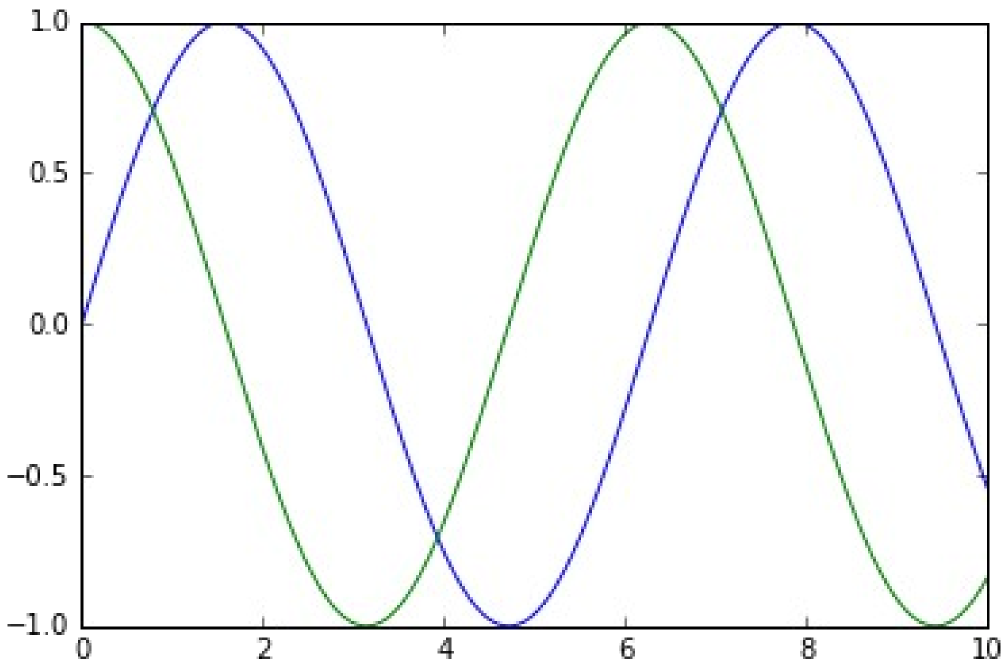
```
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.__version__
```

{"type":"string"}

```
#setting ploting style
plt.style.use('classic')
```

```
%matplotlib inline
import numpy as np
x = np.linspace(0, 10, 1000)

fig = plt.figure()
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```



Saving Figures to File

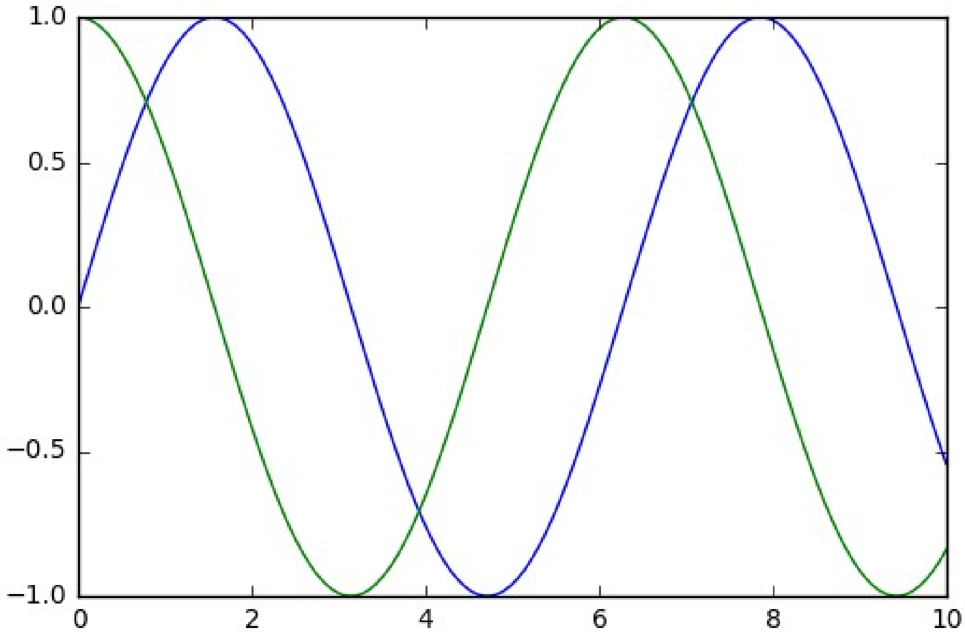
One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the `savefig()` command. For example, to save the previous figure as a PNG file, you can run this:

```
#Save above plot with file named my_figure.png in Current Working
Directory
fig.savefig('my_figure.png')

# Cross check the folder content
!ls -lh my_figure.png

-rw-r--r-- 1 root root 31K Feb 23 13:00 my_figure.png

# use Image package in python to check contents of png file in cwd
from IPython.display import Image
Image('my_figure.png')
```




```python
#checking supported file types by matplotlib
fig.canvas.get_supported_filetypes()
```

```
{'eps': 'Encapsulated Postscript',
 'jpeg': 'Joint Photographic Experts Group',
 'jpg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'ps': 'Postscript',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

Matplotlib API Interfaces

A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface. This section we will review them breifly.

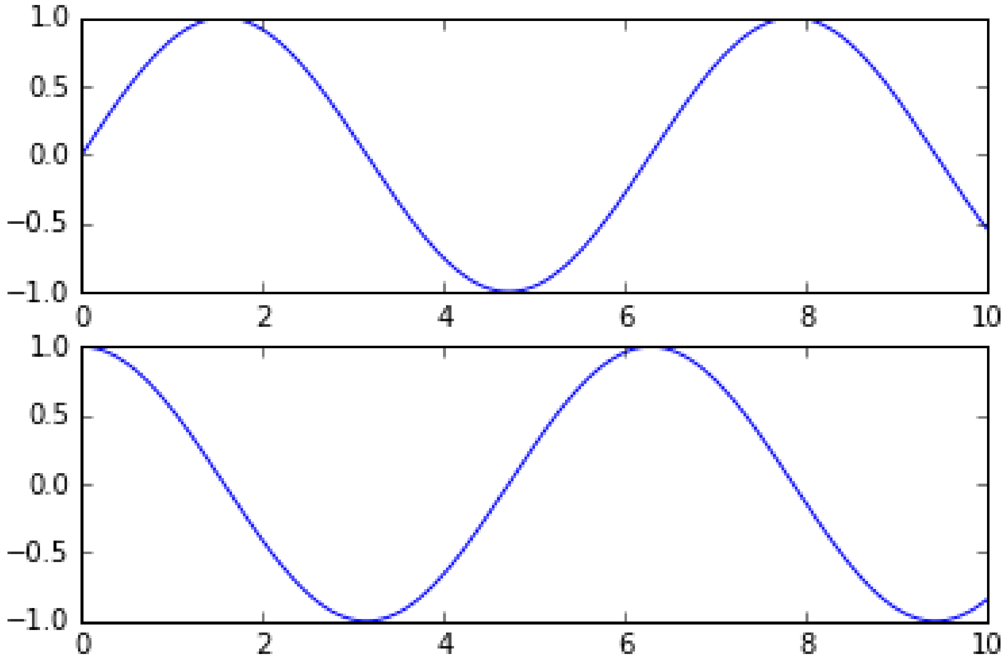
MATLAB-style Interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (`plt`) interface. See the example inthe code cell below.

```
plt.figure()  # create a plot figure

# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



This interface is *stateful*: It keeps track of the "current" figure and axes, which are where all `plt` commands are applied.

You can get a reference to these using the `plt.gcf()` (get current figure) and `plt.gca()` (get current axes) routines.

While this stateful interface is fast and convenient for simple plots, it is easy to run into problems.

For example, once the second panel is created, how can we go back and add something to the first? This interface does have a solution but it is not a easy solution.

Object-oriented interface of Matplotlib

The object-oriented interface can nicely handle more complicated plots, and for when you want more control over your figure.

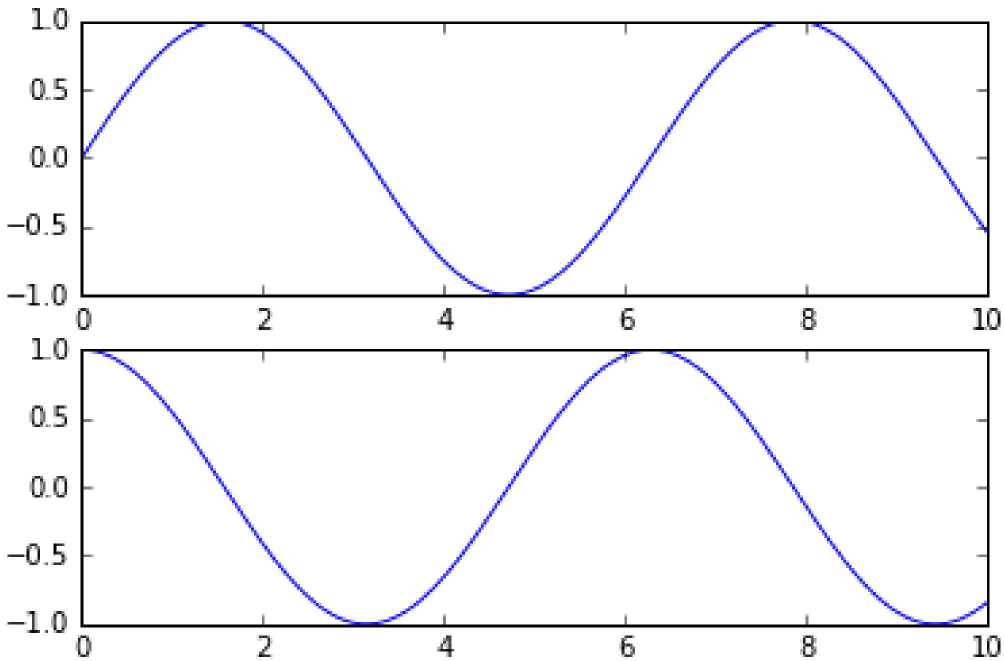
Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are *methods* of explicit `Figure` and `Axes` objects.

To re-create the previous plot using this style of plotting, you might do the following:

```
#First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))
```

```
[<matplotlib.lines.Line2D at 0x7f3a288a3b10>]
```



The difference is as small as switching `plt.plot()` to `ax.plot()`.

Now we start exploring different types of plots in following sections.

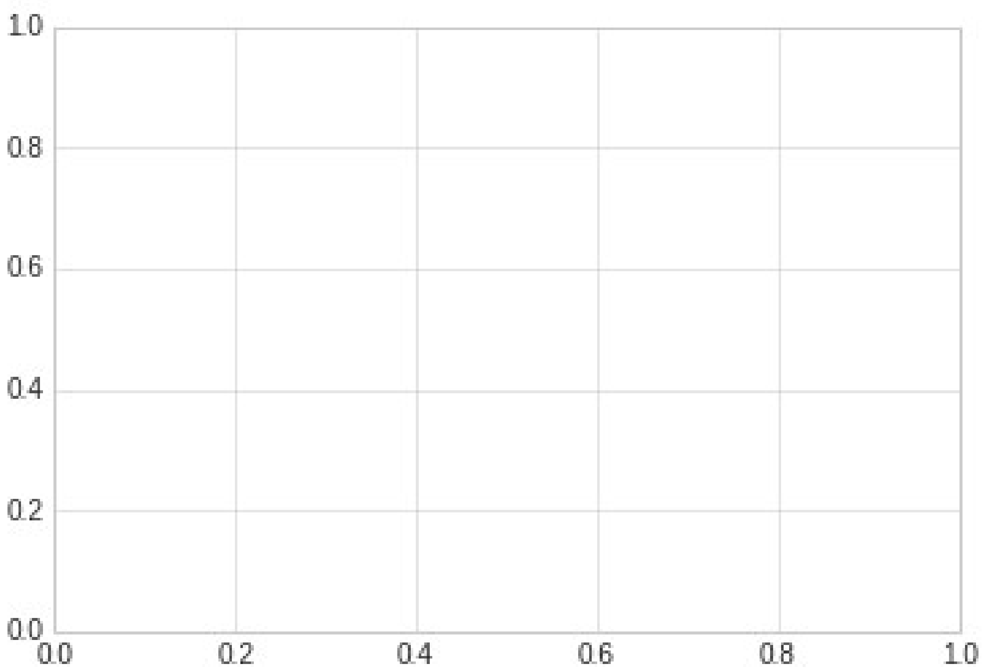
Simple Line Plots

Perhaps the simplest of all plots is the visualization of a single function `y = f(x)`.

Here we will take a first look at creating a simple plot of this type.

```
# Import all necesssary libraries
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
#ploting style chosen is that of seaborn-whitegrid
plt.style.use('seaborn-whitegrid')
```

```python
# creating objects of figure and axes.
fig = plt.figure()
ax = plt.axes()
```

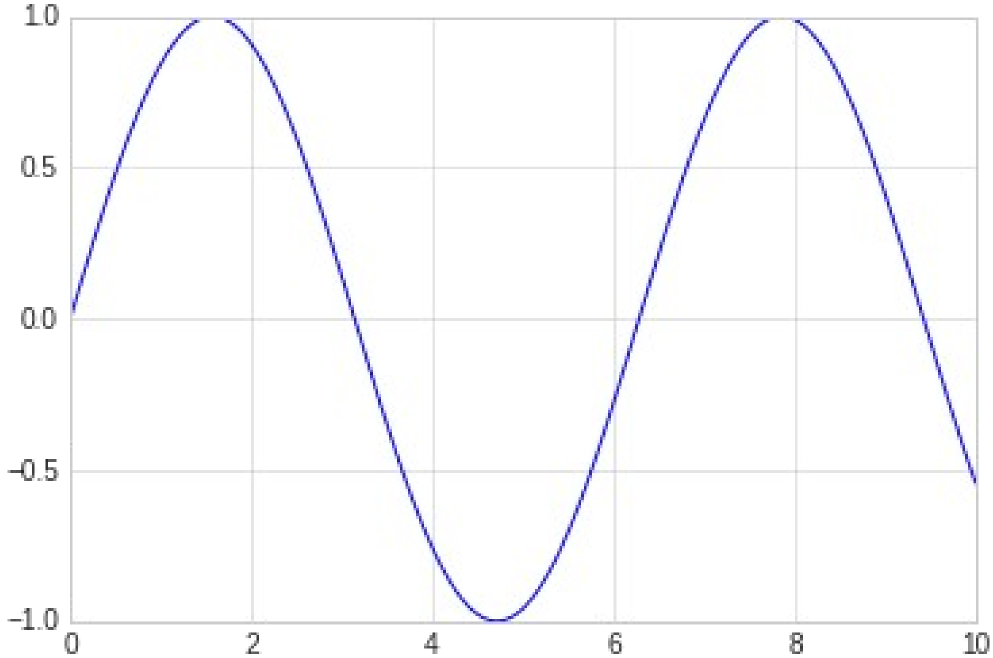


In Matplotlib, the *figure* (an instance of the class `plt.Figure`) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels.

The *axes* (an instance of the class `plt.Axes`) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.

Once we have created an axes, we can use the `ax.plot` function to plot some data.

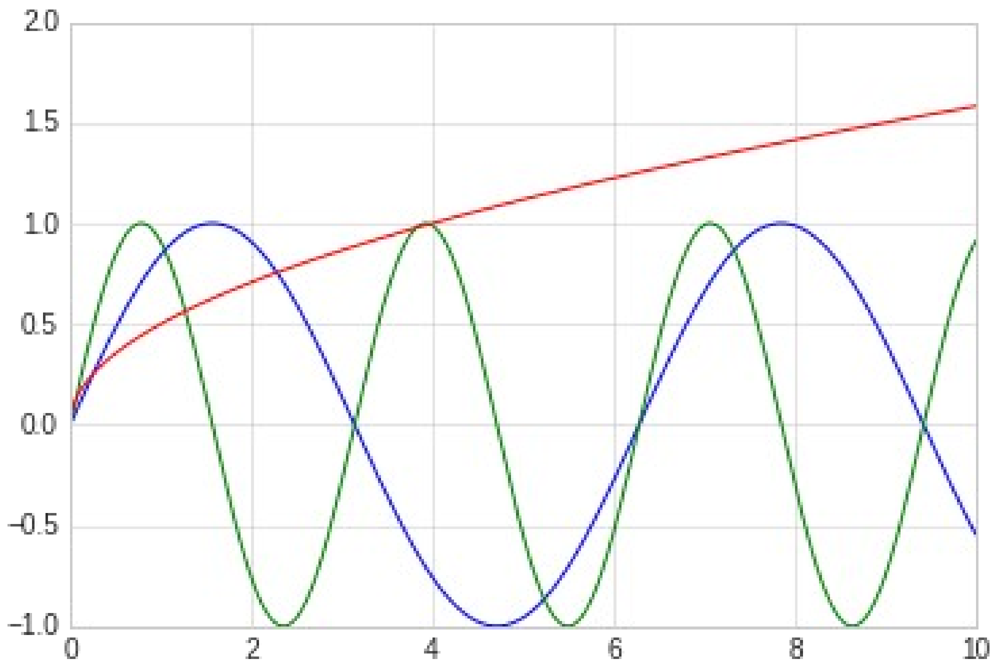
```python
# we will plot sine curve using ax.plot function
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```



The word line in line plot refers to the function we want to plot. Multiple functions can be plot on same plot like shown below.

```
plt.plot(x, np.sin(x))   #shown in blue
plt.plot(x, np.sin(2*x)) #Shown in green
plt.plot(x,(x**(1/2))/2) #Shown in red
```

```
[<matplotlib.lines.Line2D at 0x7f3a27d50d10>]
```



###Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The *plt.plot()* function takes additional arguments that can be used to specify these. To adjust the color, you can use the color keyword, which accepts a *string* argument representing virtually any imaginable color. The color can be specified in a variety of ways:

```
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by
name
plt.plot(x, np.sin(x - 1), color='g')           # short color code
(rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale between 0
and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code (RRGGBB
from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0
to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names
supported
```

```
---------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-1-e996011780af> in <module>()
----> 1 plt.plot(x, np.sin(x - 0), color='blue')        # specify
color by name
      2 plt.plot(x, np.sin(x - 1), color='g')           # short color
code (rgbcmyk)
      3 plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale
between 0 and 1
      4 plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code
(RRGGBB from 00 to FF)
      5 plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple,
values 0 to 1

NameError: name 'plt' is not defined
```

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```

###Adjusting the Plot: Axes Limits

Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust axis limits is to use the *plt.xlim()* and *plt.ylim()* methods:

```
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```

```
# Reverse the Axis Limit
plt.plot(x, np.sin(x))

plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```

A useful related method is plt.axis() (note here the potential confusion between axes with an e, and axis with an i). The plt.axis() method allows you to set the x and y limits with a single call, by passing a list which specifies **[xmin, xmax, ymin, ymax] :**

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```

###Labeling Plots

As the last piece of this section, we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.

Titles and axis labels are the simplest such labels—there are methods that can be used to quickly set them:

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```

When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type.

Again, Matplotlib has a built-in way of quickly creating such a legend. It is done via the (you guessed it) **plt.legend()** method.

Though there are several valid ways of using this, I find it easiest to specify the label of each line using the label keyword of the plot function:

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')

plt.legend();
```

In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the ax.set() method to set all these properties at once:

```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2),
       xlabel='x', ylabel='sin(x)',
       title='A Simple Plot');
```

###Simple Scatter Plots Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
             label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

###Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the plt.scatter function, which can be used very similarly to the plt.plot function:

```
plt.scatter(x, y, marker='o');
```

```
# Consider this code for scatter plot

rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis')
plt.colorbar();  # show color scale
```

##Question Answer Section

1. List any 2 **python APIs** that are used for Plotting Graphs.

-> 1)matplotlib Matplotlib is a data visualization library and 2-D plotting library of Python It was initially released in 2003 and it is the most popular and widely-used plotting library in the Python community. It comes with an interactive environment across multiple platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, etc. It can be used to embed plots into applications using various GUI toolkits like Tkinter, GTK+, wxPython, Qt, etc. So you can use Matplotlib to create plots, bar charts, pie charts, histograms, scatterplots, error charts, power spectra, stemplots, and whatever other visualization charts you want! The Pyplot module also provides a MATLAB-like interface that is just as versatile and useful as MATLAB while being free and open source. 2)Plotly Plotly is a free open-source graphing library that can be used to form data visualizations. Plotly (plotly.py) is built on top of the Plotly JavaScript library (plotly.js) and can be used to create web-based data visualizations that can be displayed in Jupyter notebooks or web applications using Dash or saved as individual HTML files. Plotly provides more than 40 unique chart types like scatter plots, histograms, line charts, bar charts, pie charts, error bars, box plots, multiple axes, sparklines, dendrograms, 3-D charts, etc. Plotly also provides contour plots, which are not that common in other data visualization libraries. In addition to all this, Plotly can be used offline with no internet connection

1. Name 2 important machnisms used for accesssing Matplotlib API.

-> 1)Pyplot: Pyplot is a collection of functions in the matplotlib module that provide a simple interface for creating a variety of plots such as line plots, scatter plots, bar plots, and histograms. It is a state-based interface and is mostly used for interactive plotting. Pyplot automatically creates and manages figures and axes, and provides a wide range of customization options. 2)Object-oriented API: The object-oriented API in Matplotlib provides full control over the figure and axes objects, and allows for more advanced customizations. It is a more lowlevel approach and is suitable for creating complex and highly customized plots. With the object-oriented API, the user creates the figure and axes objects explicitly and then calls methods on these objects to create and customize the plot.

1. Create a collection of 100 random normal numbers with parameter mean 30, sd 4 store it in object summer_temp

```
import numpy as np

import matplotlib.pyplot as plt

np.random.seed(42)

summer_temp = np.random.normal(loc=30, scale=4, size=100)

print(summer_temp[:10])

plt.hist(summer_temp, bins=20)

plt.xlabel('Temperature')

plt.ylabel('Frequency')

plt.title('Distribution of Summer Temperatures')

plt.show()
```

[31.98685661

29.4469428

32.59075415

36.09211943

29.0633865

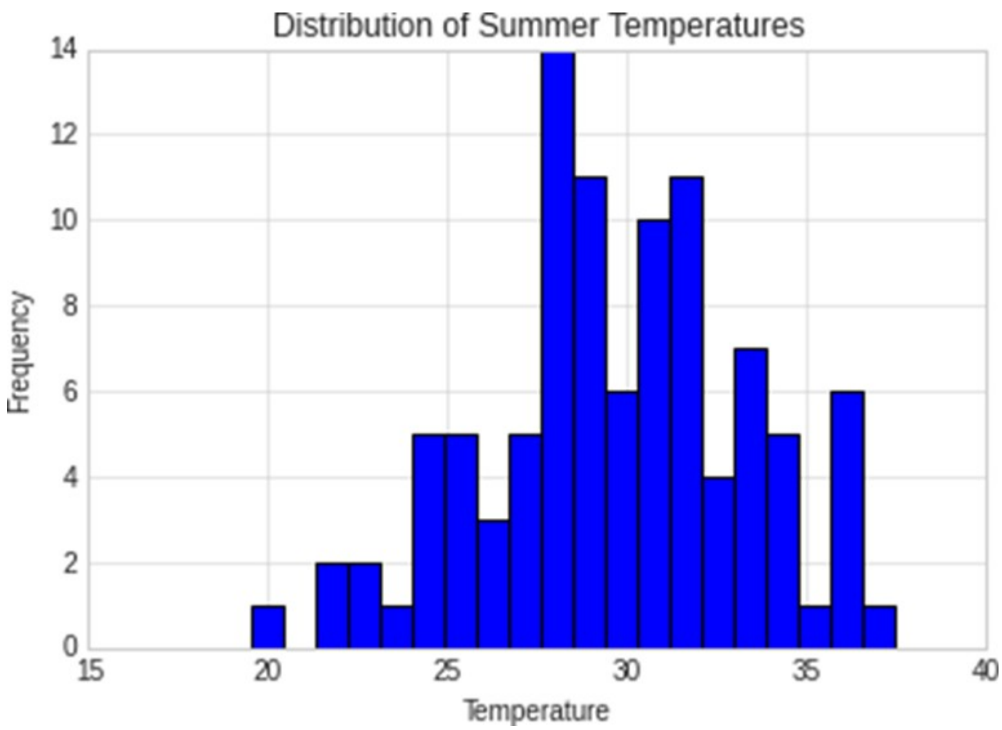
29.06345217

36.31685126

33.06973892

28.12210246

32.17024017]



1. Sort above set and store in objrct sorted_summer_temp

import numpy as np

np.random.seed(42)

summer_temp = np.random.normal(loc=30, scale=4, size=100)

sorted_summer_temp = np.sort(summer_temp)

print(sorted_summer_temp)

[19.52101958

22.04972434

22.1613195

22.34687902

22.94783938

23.10032867

24.08591204

24.14594021

24.30100726

24.35078519

24.6872558

25.1166254

25.2151735

25.39602569

25.5746601

25.76915628

25.94867552

26.3679037

26.64312991

26.76602559

27.12062317

27.19178762

27.292312

27.41952098

27.59317355

27.59744524

27.75084988

27.8224691

27.88095918

27.92691913

27.99297183

28.08330305

28.12210246

28.13708099

28.14632923

28.15744492

28.43156739

28.45967088

28.68935141

28.7631505

28.79558522

28.8039706

28.833225

29.06165147

29.0633865

29.06345217

29.0968948

29.12131245

29.25736409

29.4469428

29.53740687

29.7119595 1 29.85669584

29.9460111

30.02045383

30.27011282

30.34818827

30.36704311

30.3883102

30.44369036

30.68547312

30.78744494

30.83545438

30.96784909

31.04422109

31.18448111

31.25698933

31.29633588

31.31500444

31.32505373

31.37447316

31.42845029

31.44558242

31.4465441

31.50279207

31.98685661

32.05306973

32.17024017

32.44670516

32.59075415

32.95386632

33.06973892

33.25010329

33.28761002

33.29017965

33.66160847

33.72512048

33.87457996

33.90218051

34.01413159

34.12399809

34.2284889

35.42496011

35.86259508

35.91157618

36.09211943

36.15214627

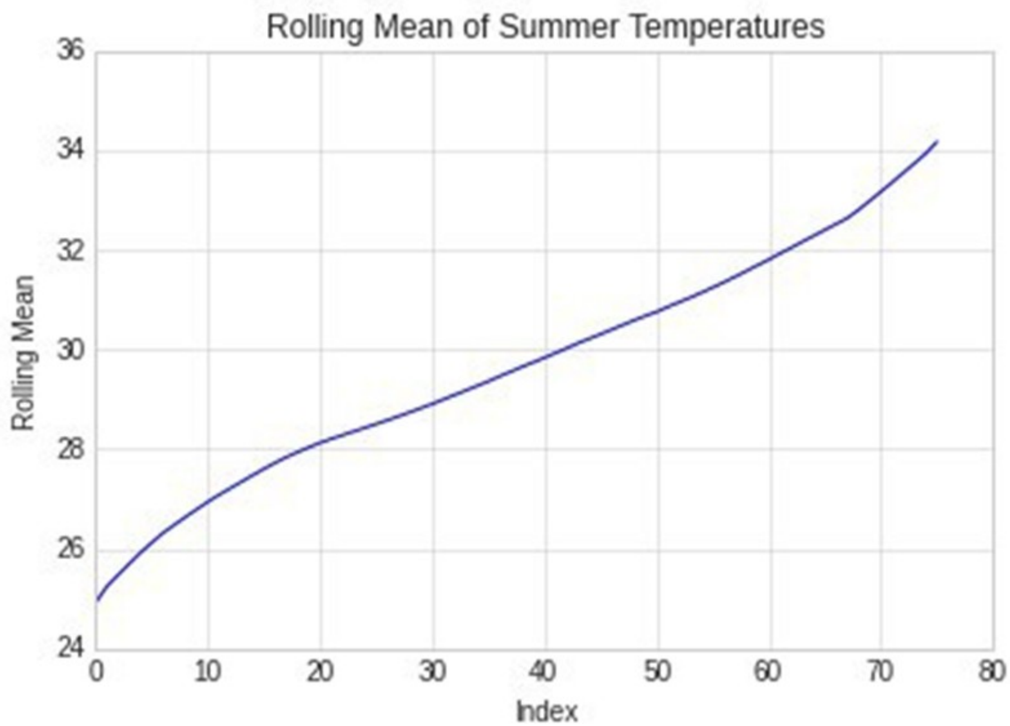
36.25857462

36.31685126

37.40911274]

1. Store average of 25 consecutive elements from sorted_summer_temp into object x and plot it as a *line plot*. What kind of information do u get about the data?

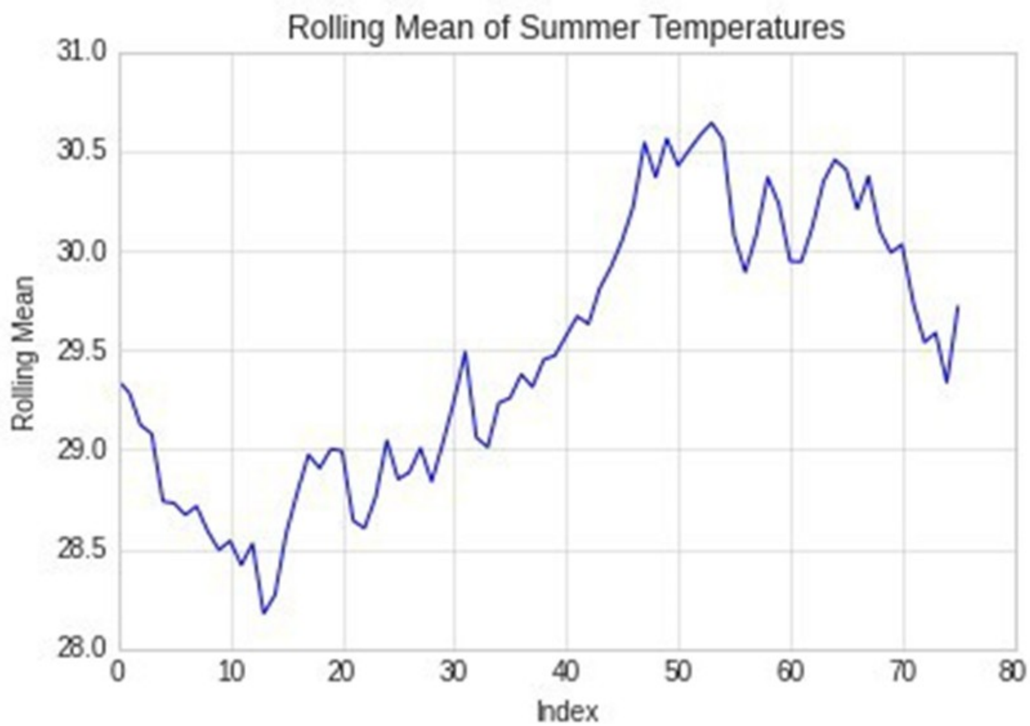
x = np.convolve(sorted_summer_temp, np.ones(25)/25, mode='valid') plt.plot(x) plt.xlabel('Index') plt.ylabel('Rolling Mean') plt.title('Rolling Mean of Summer Temperatures') plt.show()



1. Repeat above step for summer_temp, store summary in object y and this time use *scatter plot* to plot values. What kind of information do u get about the data?

y = np.convolve(summer_temp, np.ones(25)/25, mode='valid')

plt.plot(y) plt.xlabel('Index') plt.ylabel('Rolling Mean') plt.title('Rolling Mean of Summer Temperatures') plt.show()

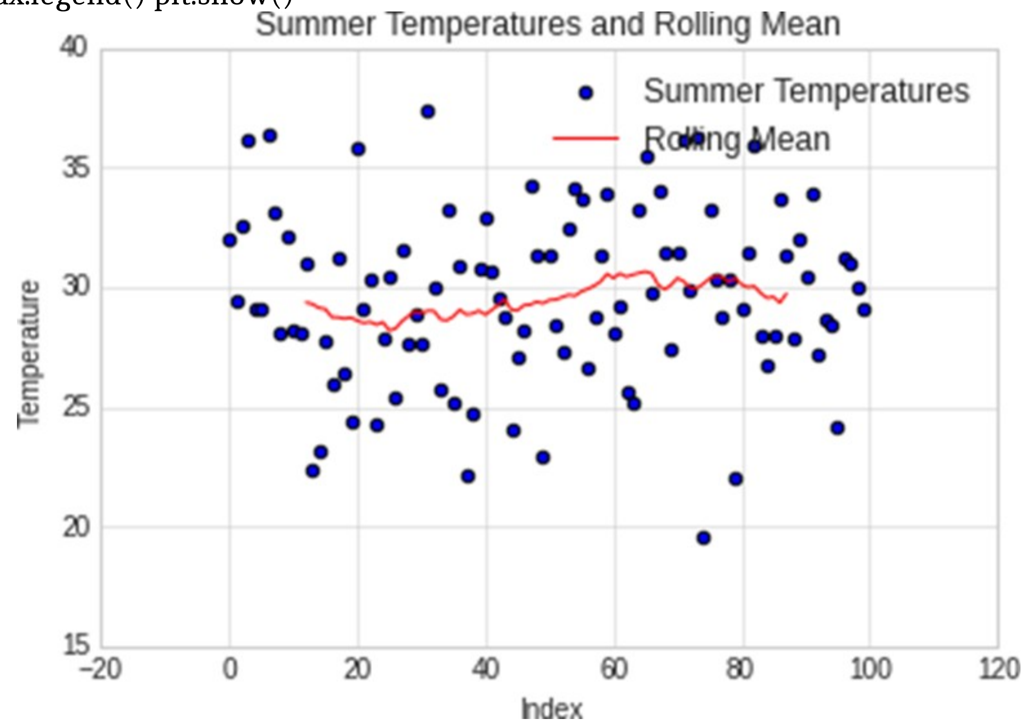


1. plot *summer_temp* and *y* together in same line plot.

fig, ax = plt.subplots()

ax.scatter(range(len(summer_temp)), summer_temp, label='Summer Temperatures')

ax.plot(range(12, len(y)+12), y, label='Rolling Mean', color='red') ax.set_xlabel('Index') ax.set_ylabel('Temperature ') ax.set_title('Summer Temperatures and Rolling Mean') ax.legend() plt.show()



Note: Answer above questions by creating apropriate code /text cells

Conclusion

Thus we have learned abou basics of ploting using *matplotlib*.