Data Visualization with Matplotlib

This project is all about Matplotlib, the basic data visualization tool of Python programming language. I have discussed Matplotlib object hierarchy, various plot types with Matplotlib and customization techniques associated with Matplotlib.

This project is divided into various sections based on contents which are listed below:-

Table of Contents

1. Introduction

When we want to convey some information to others, there are several ways to do so. The process of conveying the information with the help of plots and graphics is called **Data Visualization**. The plots and graphics take numerical data as input and display output in the form of charts, figures and tables. It helps to analyze and visualize the data clearly and make concrete decisions. It makes complex data more accessible and understandable. The goal of data visualization is to communicate information in a clear and efficient manner.

In this project, I shed some light on **Matplotlib**, which is the basic data visualization tool of Python programming language. Python has different data visualization tools available which are suitable for different purposes. First of all, I will list these data visualization tools and then I will discuss Matplotlib.

2. Overview of Python Visualization Tools

Python is the preferred language of choice for data scientists. Python have multiple options for data visualization. It has several tools which can help us to visualize the data more effectively. These Python data visualization tools are as follows:-

• Matplotlib

• Seaborn

• pandas

• Bokeh

• Plotly

• ggplot

• pygal

In the following sections, I discuss Matplotlib as the data visualization tool.

3. Introduction to Matplotlib

Matplotlib is the basic plotting library of Python programming language. It is the most prominent tool among Python visualization packages. Matplotlib is highly efficient in performing wide range of tasks. It can produce publication quality figures in a variety of formats. It can export visualizations to all of the common formats like PDF, SVG, JPG, PNG, BMP and GIF. It can create popular visualization types – line plot, scatter plot, histogram, bar chart, error charts, pie chart, box plot, and many more types of plot. Matplotlib also supports 3D plotting. Many Python libraries are built on top of Matplotlib. For example, pandas and Seaborn are built on Matplotlib. They allow to access Matplotlib's methods with less code.

The project **Matplotlib** was started by John Hunter in 2002. Matplotlib was originally started to visualize Electrocorticography (ECoG) data of epilepsy patients during post-doctoral research in Neurobiology. The open-source tool Matplotlib emerged as the most widely used plotting library for the Python programming language. It was used for data visualization during landing of the Phoenix spacecraft in 2008.

4. Import Matplotlib

Before, we need to actually start using Matplotlib, we need to import it. We can import Matplotlib as follows:-

```
import matplotlib
```

Most of the time, we have to work with **pyplot** interface of Matplotlib. So, I will import **pyplot** interface of Matplotlib as follows:-

```
import matplotlib.pyplot
```

To make things even simpler, we will use standard shorthand for Matplotlib imports as follows:-

```
import matplotlib.pyplot as plt
```

```
# Import dependencies

import numpy as np
import pandas as pd

# Import Matplotlib

import matplotlib.pyplot as plt
```

5. Displaying Plots in Matplotlib

Viewing the Matplotlib plot is context based. The best usage of Matplotlib differs depending on how we are using it. There are three applicable contexts for viewing the plots. The three applicable contexts are using plotting from a script, plotting from an IPython shell or plotting from a Jupyter notebook.

Plotting from a script

If we are using Matplotlib from within a script, then the **plt.show()** command is of great use. It starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display the figure or figures.

The **plt.show()** command should be used only once per Python session. It should be used only at the end of the script. Multiple **plt.show()** commands can lead to unpredictable results and should mostly be avoided.

Plotting from an IPython shell

We can use Matplotlib interactively within an IPython shell. IPython works well with Matplotlib if we specify Matplotlib mode. To enable this mode, we can use the **%matplotlib** magic command after starting ipython. Any plt plot command will cause a figure window to open and further commands can be run to update the plot.

Plotting from a Jupyter notebook

The Jupyter Notebook (formerly known as the IPython Notebook) is a data analysis and visualization tool that provides multiple tools under one roof. It provides code execution, graphical plots, rich text and media display, mathematics formula and much more facilities into a single executable document.

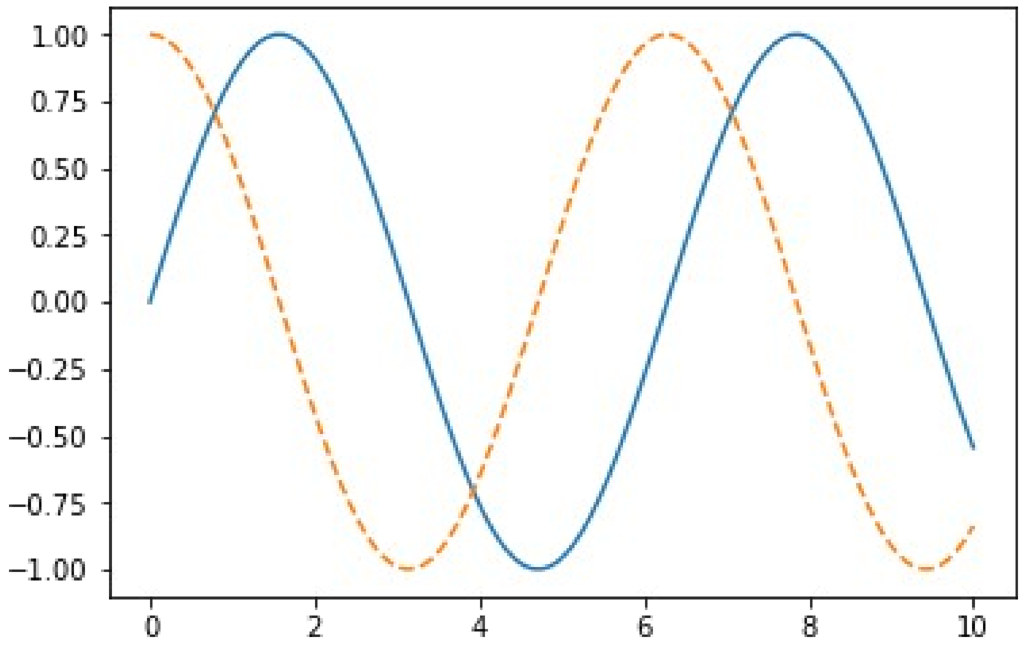
Interactive plotting within a Jupyter Notebook can be done with the **%matplotlib** command. There are two possible options to work with graphics in Jupyter Notebook. These are as follows:-

• **%matplotlib notebook** – This command will produce interactive plots embedded within the notebook.

• **%matplotlib inline** – It will output static images of the plot embedded in the notebook.

After this command (it needs to be done only once per kernel per session), any cell within the notebook that creates a plot will embed a PNG image of the graphic.

```
%matplotlib inline


x1 = np.linspace(0, 10, 100)


# create a plot figure
fig = plt.figure()

plt.plot(x1, np.sin(x1), '-')
plt.plot(x1, np.cos(x1), '--');
```



6. Matplotlib Object Hierarchy

There is an Object Hierarchy within Matplotlib. In Matplotlib, a plot is a hierarchy of nested Python objects. A**hierarch** means that there is a tree-like structure of Matplotlib objects underlying each plot.

A **Figure** object is the outermost container for a Matplotlib plot. The **Figure** object contain multiple **Axes** objects. So, the **Figure** is the final graphic that may contain one or more **Axes**. The **Axes** represent an individual plot.

So, we can think of the **Figure** object as a box-like container containing one or more **Axes**. The **Axes** object contain smaller objects such as tick marks, lines, legends, title and text-boxes.

7. Matplotlib API Overview

Matplotlib has two APIs to work with. A MATLAB-style state-based interface and a more powerful object-oriented (OO) interface. The former MATLAB-style state-based interface is called **pyplot interface** and the latter is called **Object-Oriented** interface.

There is a third interface also called **pylab** interface. It merges pyplot (for plotting) and NumPy (for mathematical functions) together in an environment closer to MATLAB. This is considered bad practice nowadays. So, the use of **pylab** is strongly discouraged and hence, I will not discuss it any further.

8. Pyplot API

Matplotlib.pyplot provides a MATLAB-style, procedural, state-machine interface to the underlying object-oriented library in Matplotlib. **Pyplot** is a collection of command style functions that make Matplotlib work like MATLAB. Each pyplot function makes some change to a figure - e.g., creates a figure, creates a plotting area in a figure etc.

Matplotlib.pyplot is stateful because the underlying engine keeps track of the current figure and plotting area information and plotting functions change that information. To make it clearer, we did not use any object references during our plotting we just issued a pyplot command, and the changes appeared in the figure.

We can get a reference to the current figure and axes using the following commands-

`plt.gcf ( )` # get current figure

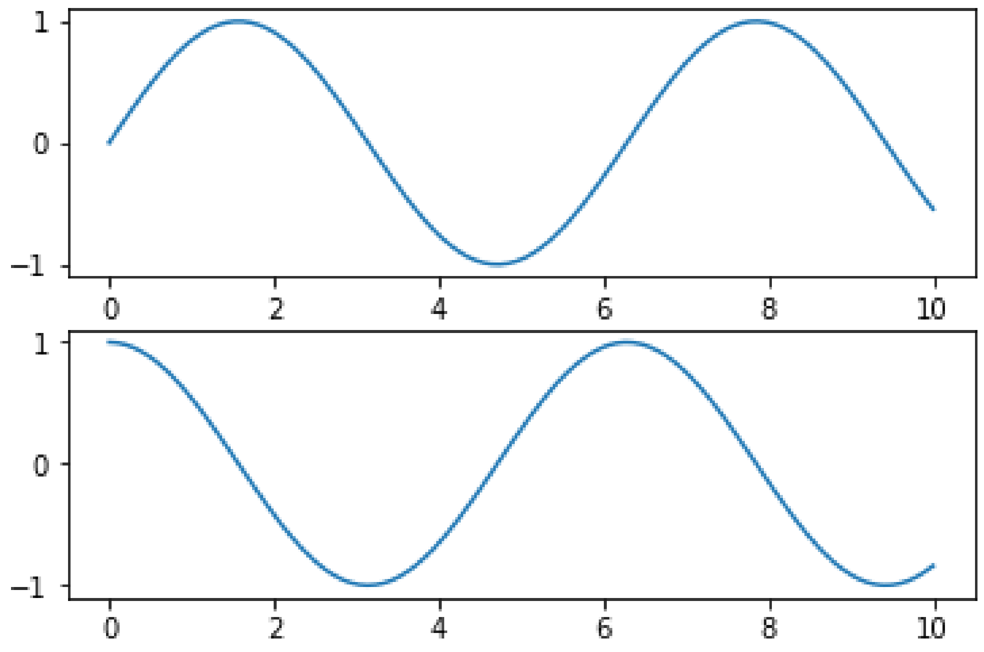
`plt.gca ( )` # get current axes

Matplotlib.pyplot is a collection of commands and functions that make Matplotlib behave like MATLAB (for plotting). The MATLAB-style tools are contained in the pyplot (plt) interface.

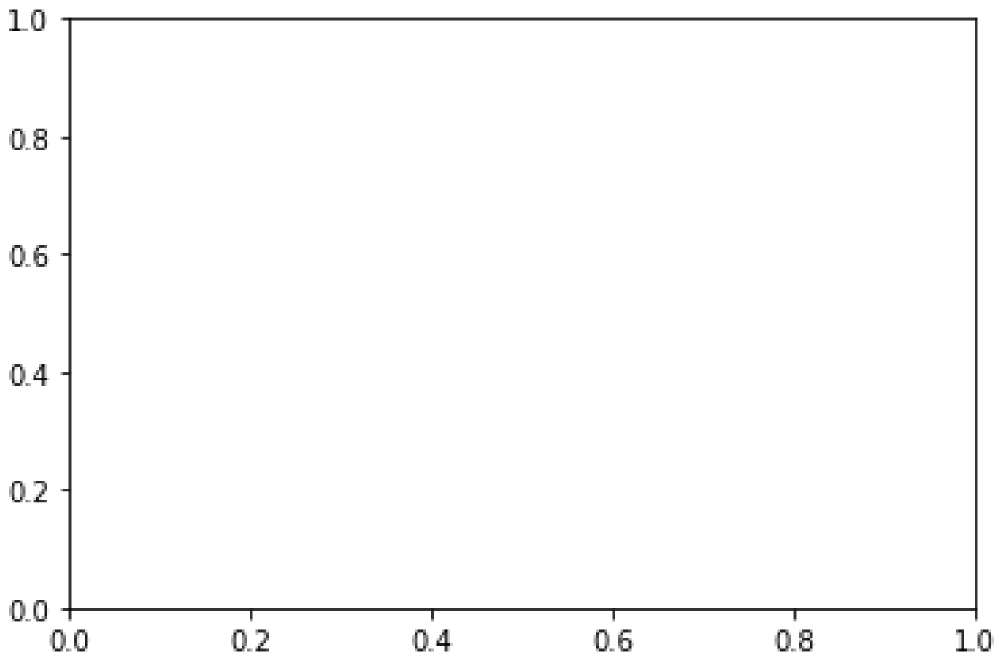
This is really helpful for interactive plotting, because we can issue a command and see the result immediately. But, it is not suitable for more complicated cases. For these cases, we have another interface called **Object-Oriented** interface, described later.

The following code produces sine and cosine curves using Pyplot API.

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


# create the first of two panels and set current axis
plt.subplot(2, 1, 1)   # (rows, columns, panel number)
plt.plot(x1, np.sin(x1))


# create the second of two panels and set current axis
plt.subplot(2, 1, 2)   # (rows, columns, panel number)
plt.plot(x1, np.cos(x1));
```



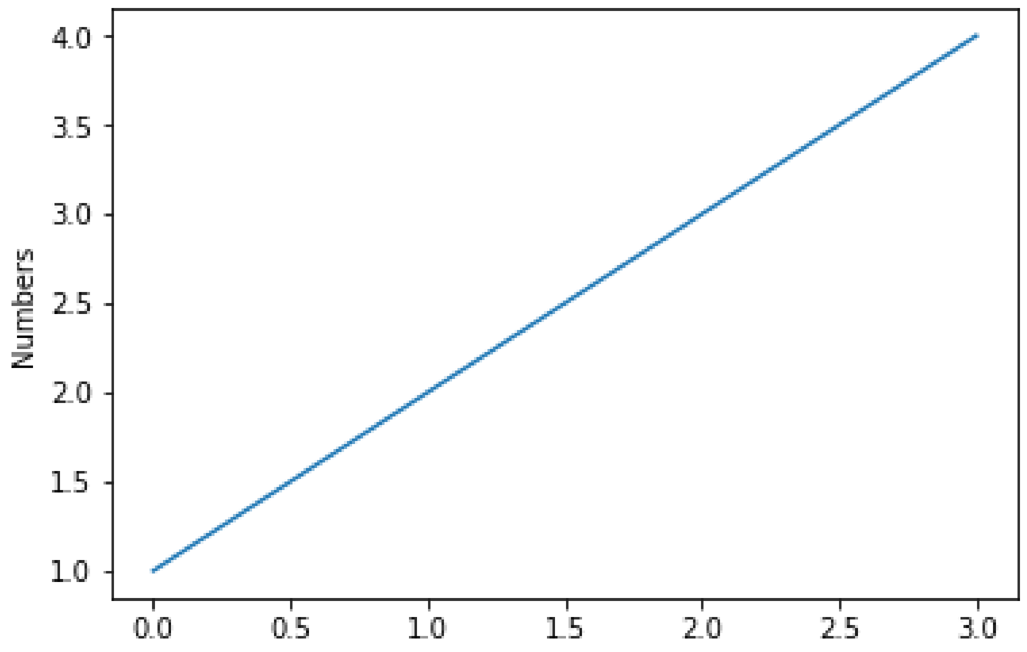
```
# get current figure information

print(plt.gcf())

Figure(432x288)

<Figure size 432x288 with 0 Axes>

# get current axis information

print(plt.gca())

AxesSubplot(0.125,0.125;0.775x0.755)
```



Visualization with Pyplot

Generating visualization with Pyplot is very easy. The x-axis values ranges from 0-3 and the y-axis from 1-4. If we provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3] and y data are [1,2,3,4].

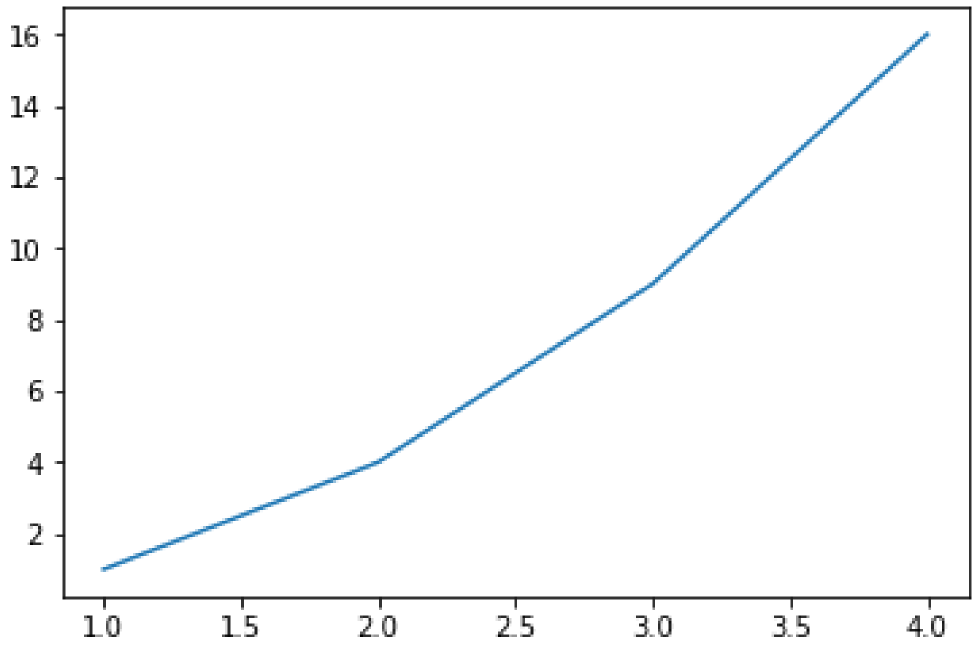
```
plt.plot([1, 2, 3, 4])
plt.ylabel('Numbers')
plt.show()
```



plot() - A versatile command

plot() is a versatile command. It will take an arbitrary number of arguments. For example, to plot x versus y, we can issue the following command:-

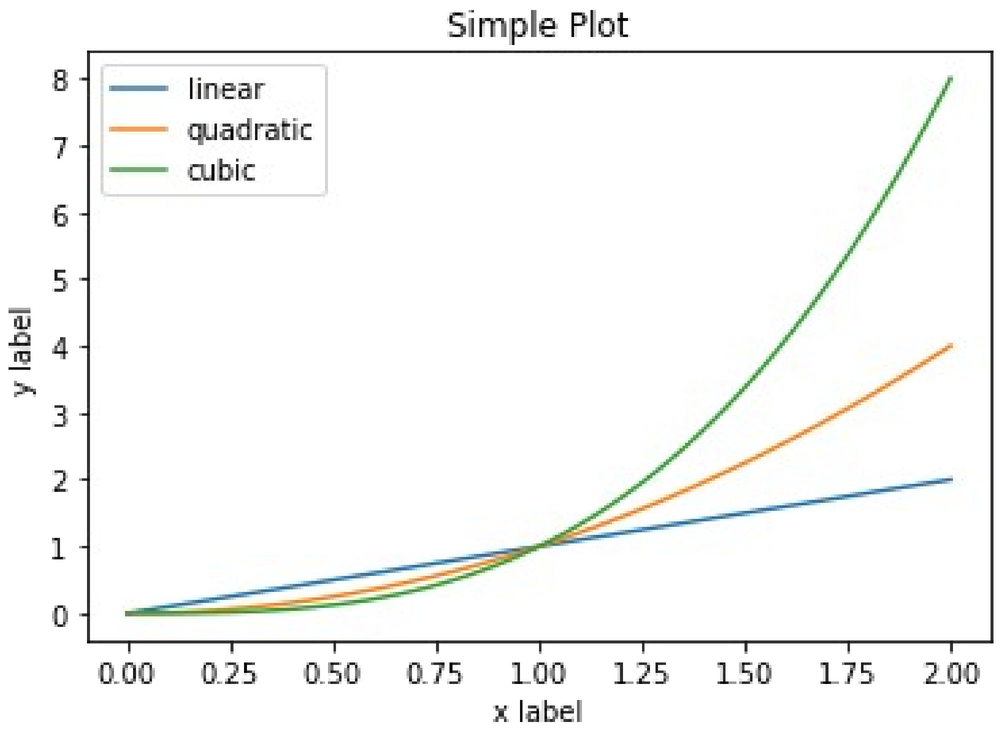
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.show()
```



State-machine interface

Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot. For example:

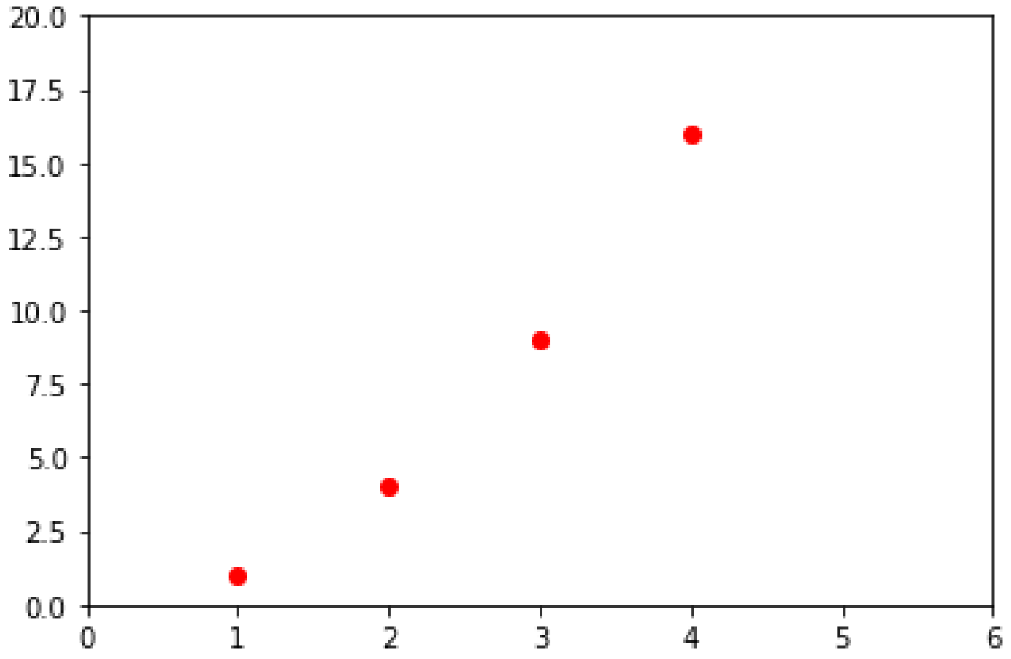
```
x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```



Formatting the style of plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB. We can concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above line with red circles, we would issue the following command:-

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

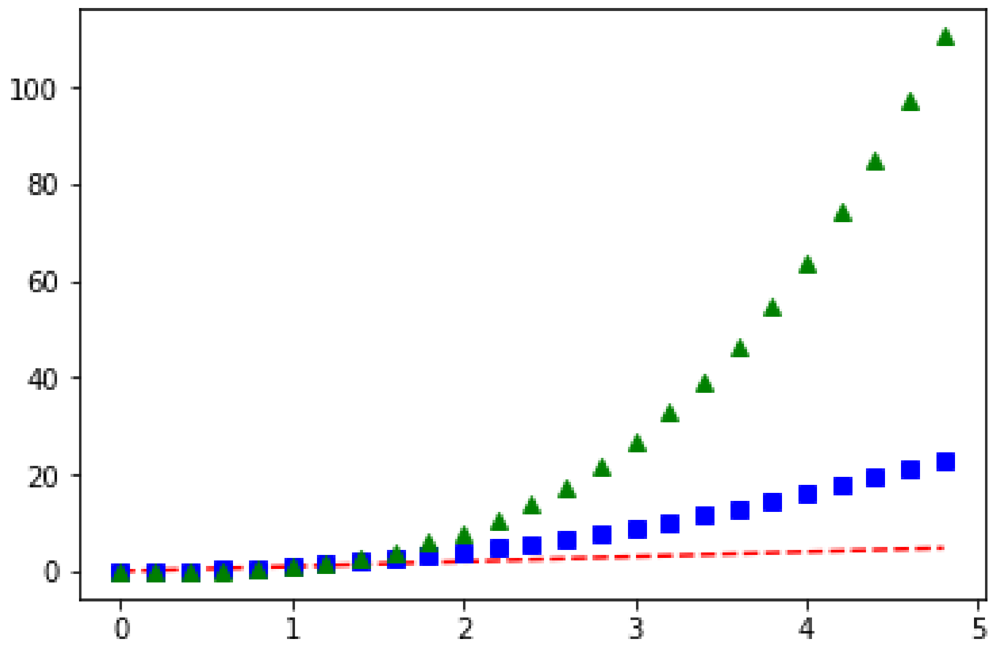


The **axis()** command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

Working with NumPy arrays

Generally, we have to work with NumPy arrays. All sequences are converted to numpy arrays internally. The below example illustrates plotting several lines with different format styles in one command using arrays.

```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



9. Object-Oriented API

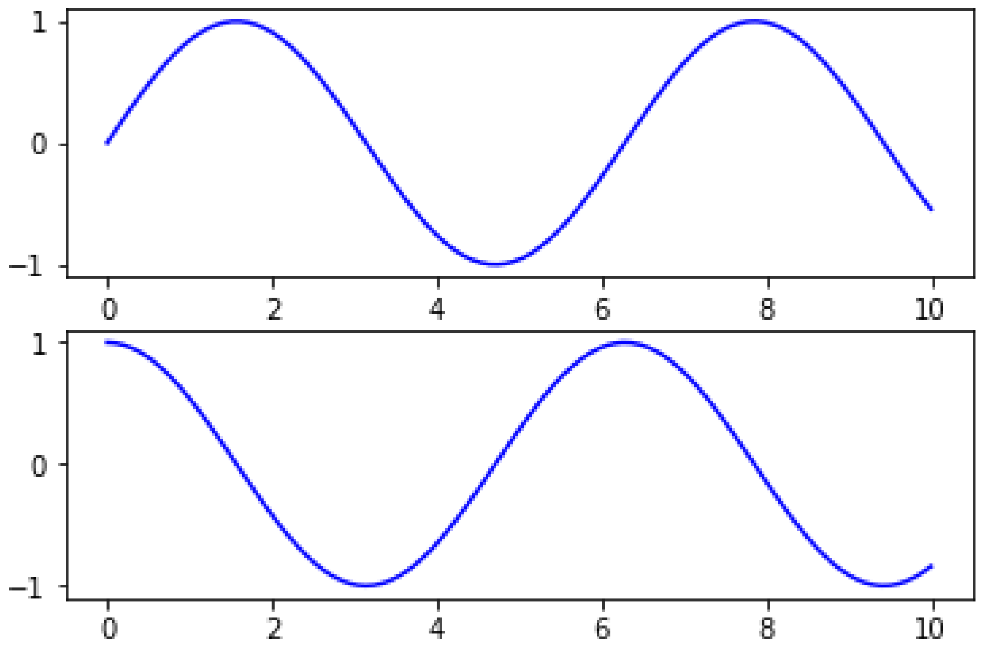
The **Object-Oriented API** is available for more complex plotting situations. It allows us to exercise more control over the figure. In Pyplot API, we depend on some notion of an "active" figure or axes. But, in the **Object-Oriented API** the plotting functions are methods of explicit Figure and Axes objects.

Figure is the top level container for all the plot elements. We can think of the **Figure** object as a box-like container containing one or more **Axes**.

The **Axes** represent an individual plot. The **Axes** object contain smaller objects such as axis, tick marks, lines, legends, title and text-boxes.

The following code produces sine and cosine curves using Object-Oriented API.

```
# 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(x1, np.sin(x1), 'b-')
ax[1].plot(x1, np.cos(x1), 'b-');
```



Objects and Reference

The main idea with the **Object Oriented API** is to have objects that one can apply functions and actions on. The real advantage of this approach becomes apparent when more than one figure is created or when a figure contains more than one subplot.

We create a reference to the figure instance in the **fig** variable. Then, we ceate a new axis instance **axes** using the **add_axes** method in the Figure class instance fig as follows:-

```
fig = plt.figure()

x2 = np.linspace(0, 5, 10)
y2 = x2 ** 2

axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])

axes.plot(x2, y2, 'r')

axes.set_xlabel('x2')
axes.set_ylabel('y2')
axes.set_title('title');
```

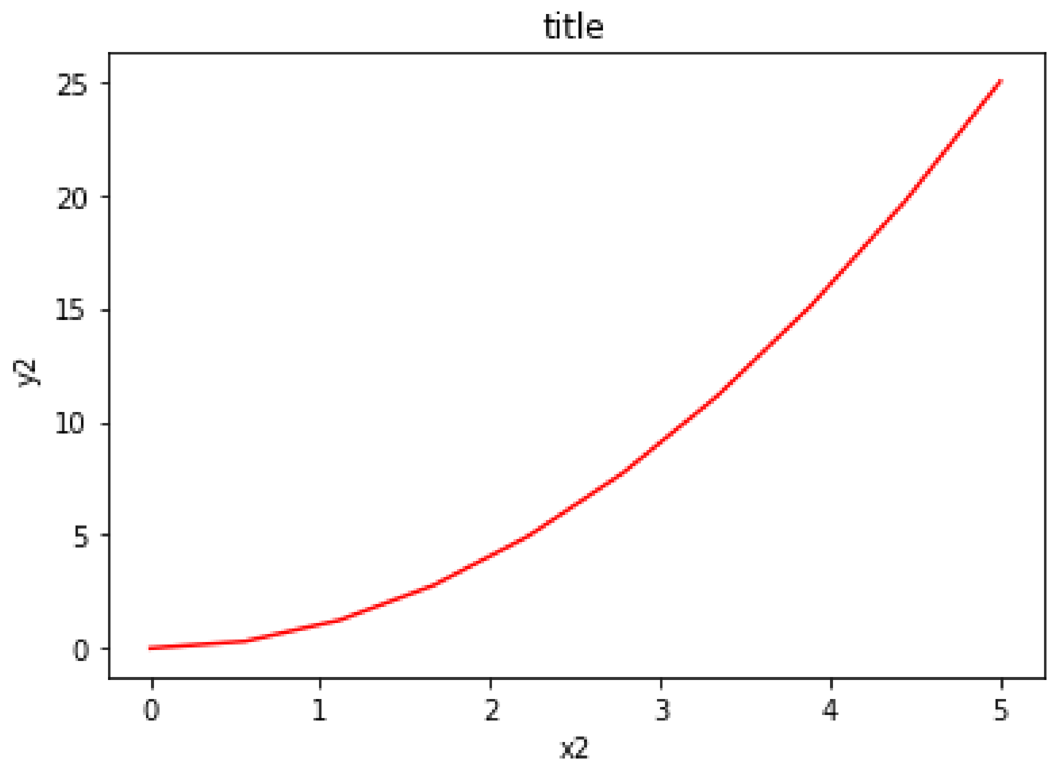


Figure and Axes

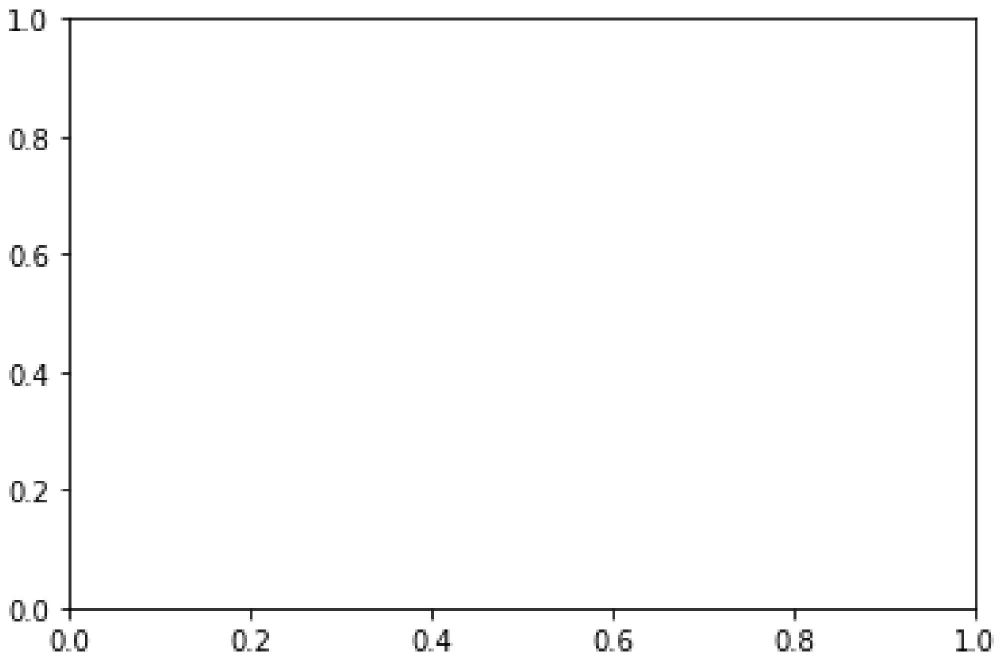
I start by creating a figure and an axes. A figure and axes can be created as follows:

`fig = plt.figure()`

`ax = plt.axes()`

In Matplotlib, the **figure** (an instance of the class plt.Figure) is a single container that contains all the objects representing axes, graphics, text and labels. The **axes** (an instance of the class plt.Axes) is a bounding box with ticks and labels. It will contain the plot elements that make up the visualization. I have used the variable name fig to refer to a figure instance, and ax to refer to an axes instance or group of axes instances.

```
fig = plt.figure()

ax = plt.axes()
```



10. Figure and Subplots

Plots in Matplotlib reside within a Figure object. As described earlier, we can create a new figure with plt.figure() as follows:-

```
fig = plt.figure()
```

Now, I create one or more subplots using fig.add_subplot() as follows:-

```
ax1 = fig.add_subplot(2, 2, 1)
```

The above command means that there are four plots in total (2 * 2 = 4). I select the first of four subplots (numbered from 1).

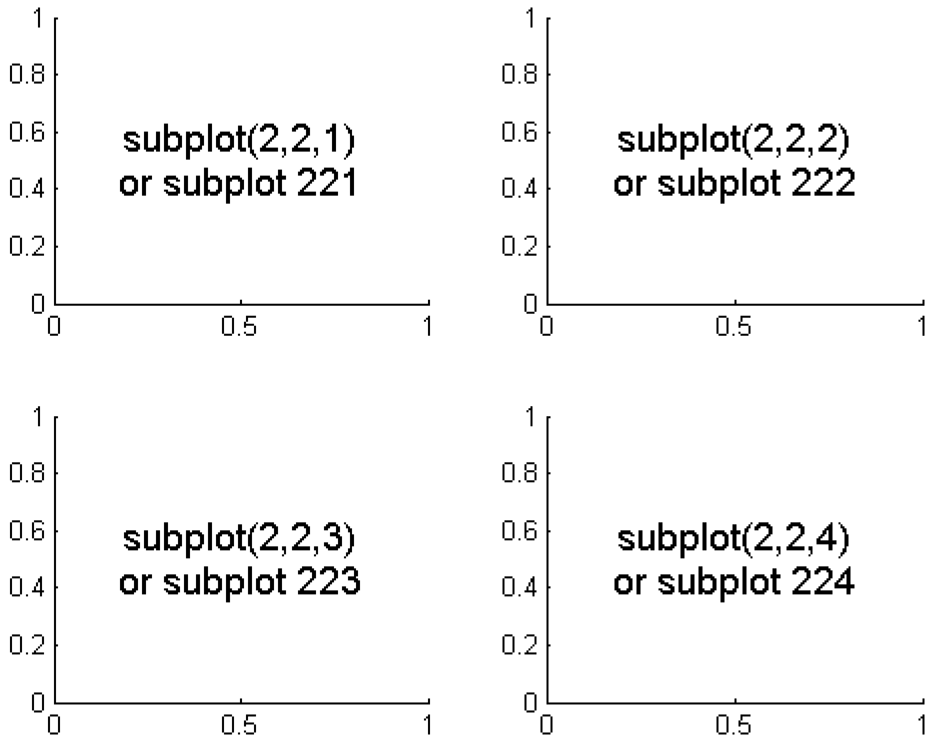
I create the next three subplots using the fig.add_subplot() commands as follows:-

```
ax2 = fig.add_subplot(2, 2, 2)
```

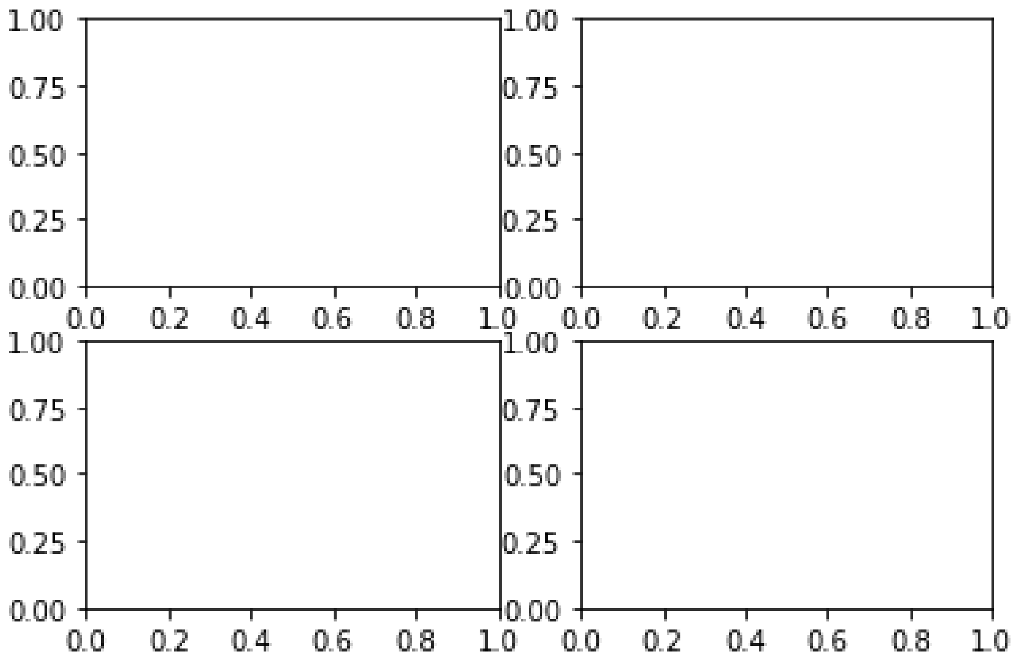
```
ax3 = fig.add_subplot(2, 2, 3)
```

```
ax4 = fig.add_subplot(2, 2, 4)
```

The above command result in creation of subplots. The diagrammatic representation of subplots are as follows:-



```
# Creating empty matplotlib figure with four subplots

import matplotlib.pyplot as plt

fig = plt.figure()

ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
ax4 = fig.add_subplot(2, 2, 4)
```



Concise representation of Subplots

There is a concise form by which we can represent Subplots. Matplotlib includes a convenience method plt.subplots() that creates a new figure and returns the subplot objects. We can create subplots as follows:-

`fig, axes = plt.subplots()`

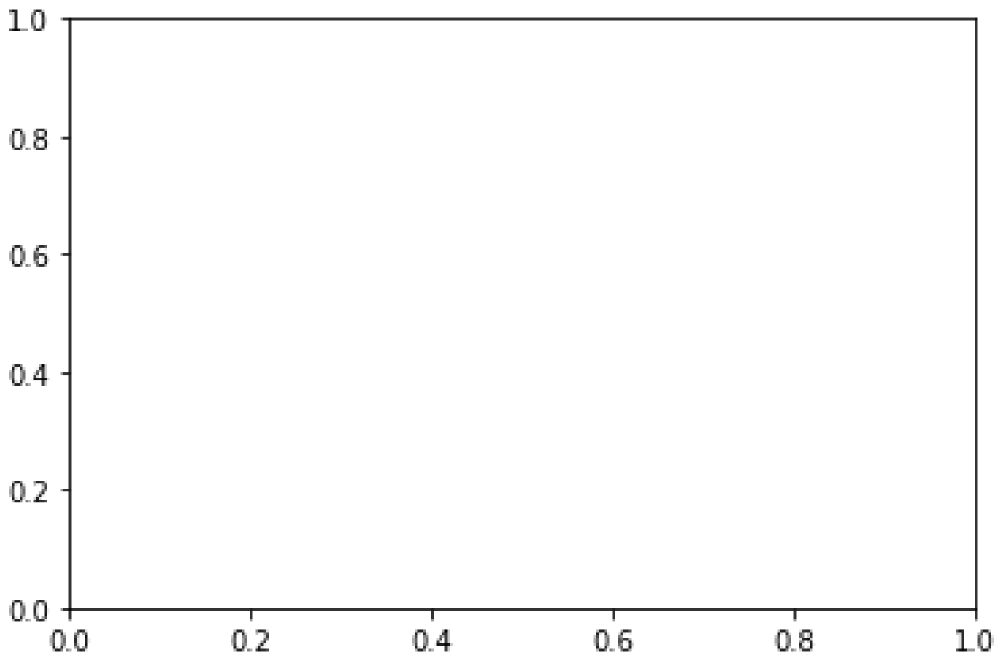
The above command creates a figure and one subplot. It is short and concise representation of the following commands:-

`fig = plt.figure()`

`ax1 = fig.add_subplot(1, 1, 1)`

We get a reference to both figure and axis in a single step.

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



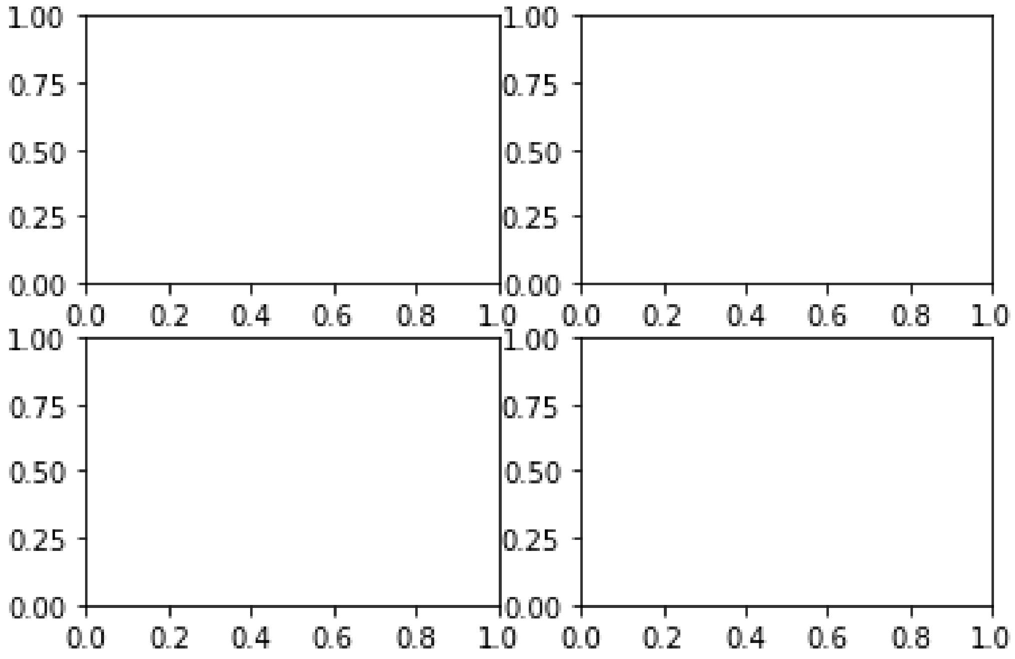
More than one Subplot

We can use

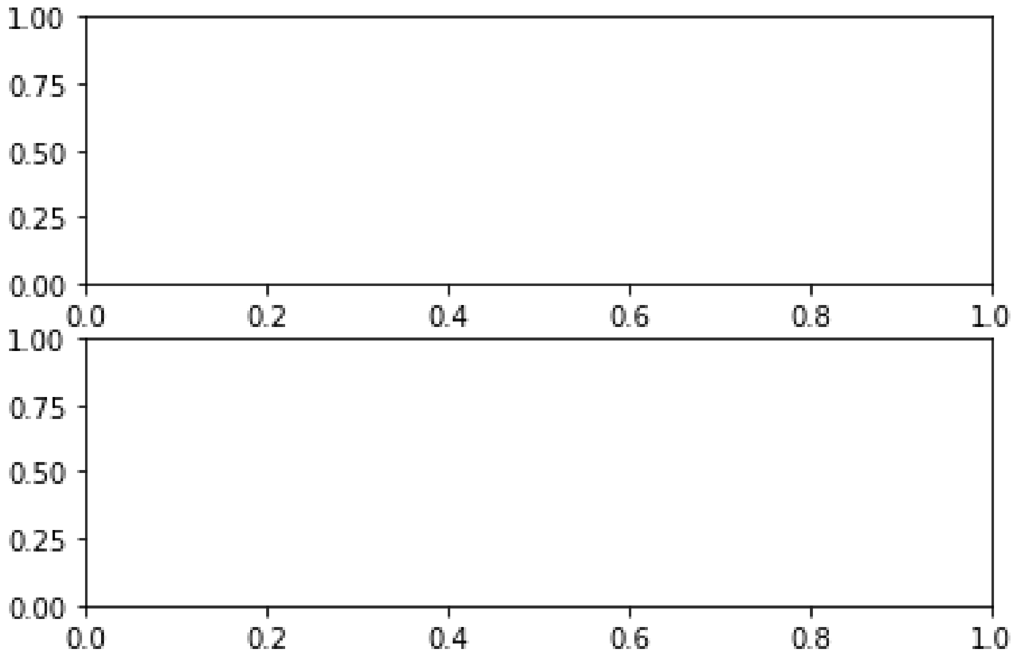
`fig, axes = plt.subplots(nrows = 2, ncols = 2)`

to create four Subplots with grid(2, 2) in one figure object.

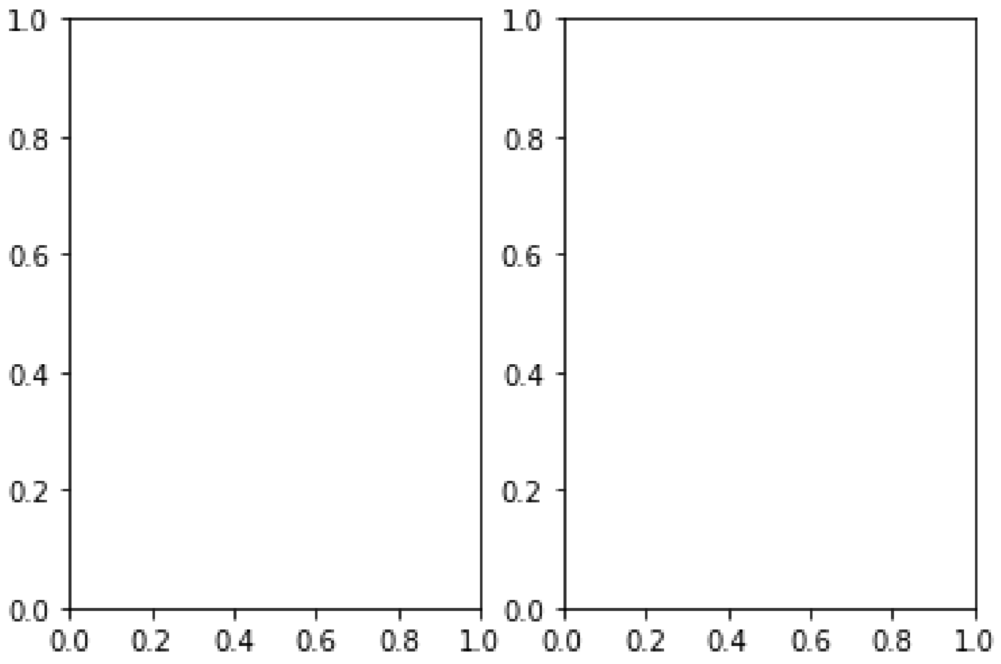
```
# Create a subplot with 2 rows and 2 columns

fig, axes = plt.subplots(nrows = 2, ncols = 2)
```



```
# Create a subplot with 2 rows and 1 column

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



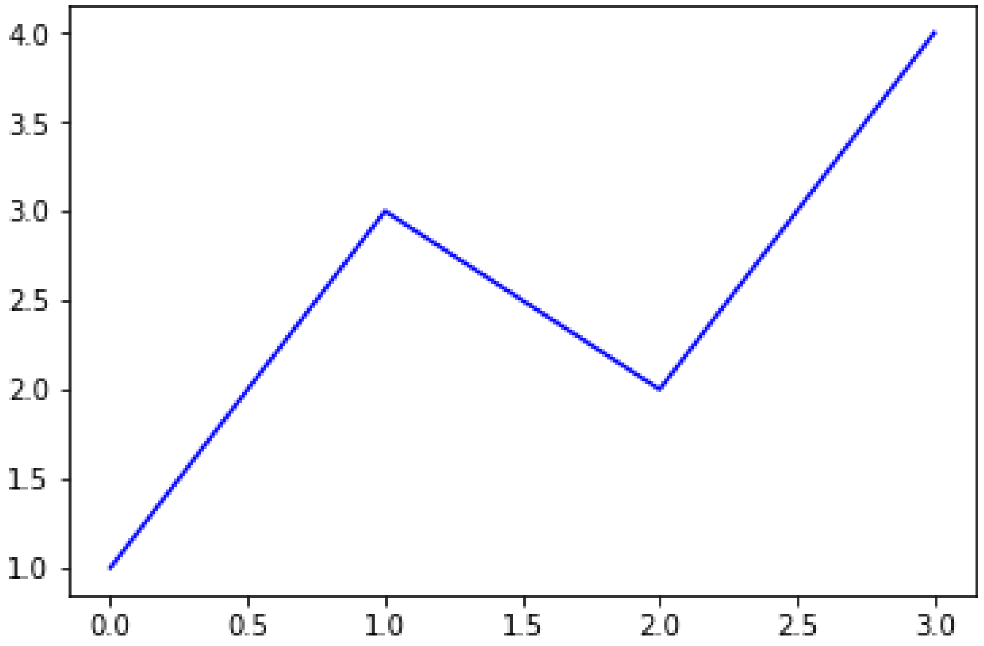
```
# Create a subplot with 1 row and 2 columns

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



11. First plot with Matplotlib

Now, I will start producing plots. Here is the first example:-

```
plt.plot([1, 3, 2, 4], 'b-')

plt.show( )
```



`plt.plot([1, 3, 2, 4], 'b-')`

This code line is the actual plotting command. Only a list of values has been plotted that represent the vertical coordinates of the points to be plotted. Matplotlib will use an implicit horizontal values list, from 0 (the first value) to N-1 (where N is the number of items in the list).

Specify both Lists

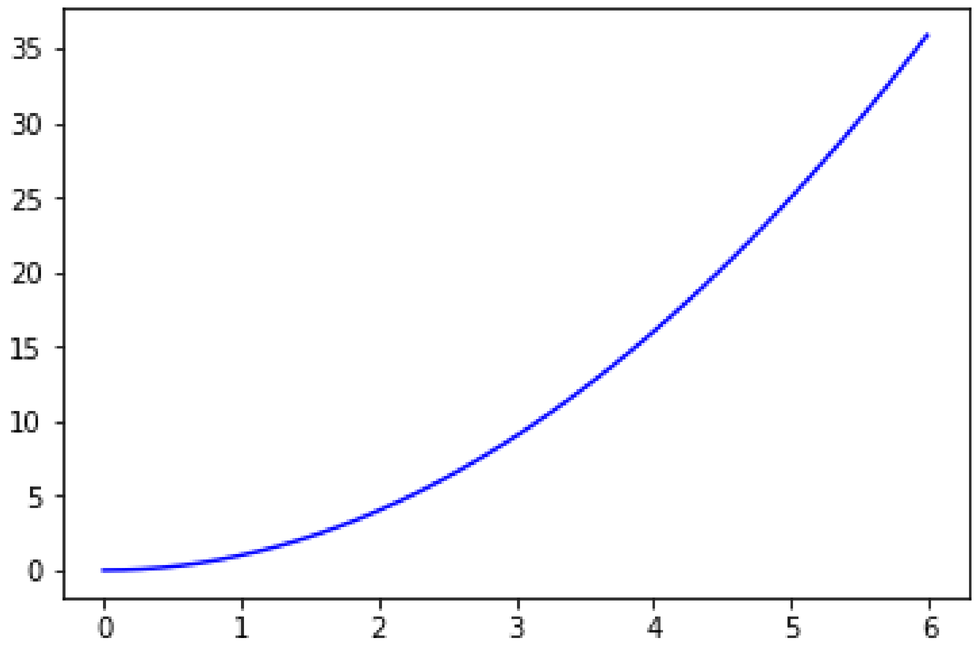
Also, we can explicitly specify both the lists as follows:-

`x3 = range(6)`

`plt.plot(x3, [xi**2 for xi in x3])`

`plt.show()`

```
x3 = np.arange(0.0, 6.0, 0.01)

plt.plot(x3, [xi**2 for xi in x3], 'b-')

plt.show()
```

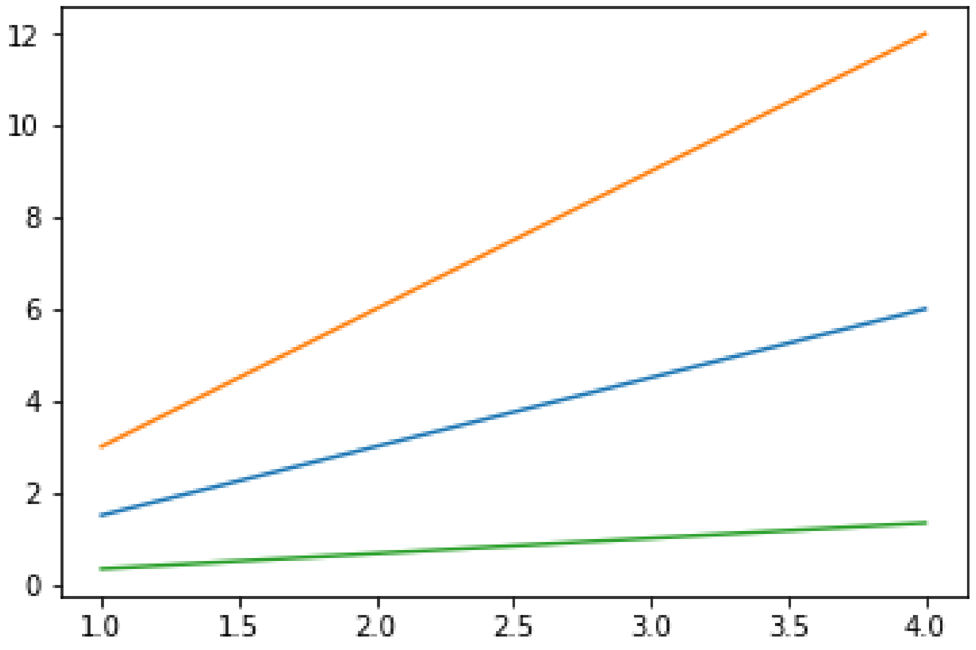


12. Multiline Plots

Multiline Plots mean plotting more than one plot on the same figure. We can plot more than one plot on the same figure.
It can be achieved by plotting all the lines before calling show(). It can be done as follows:-

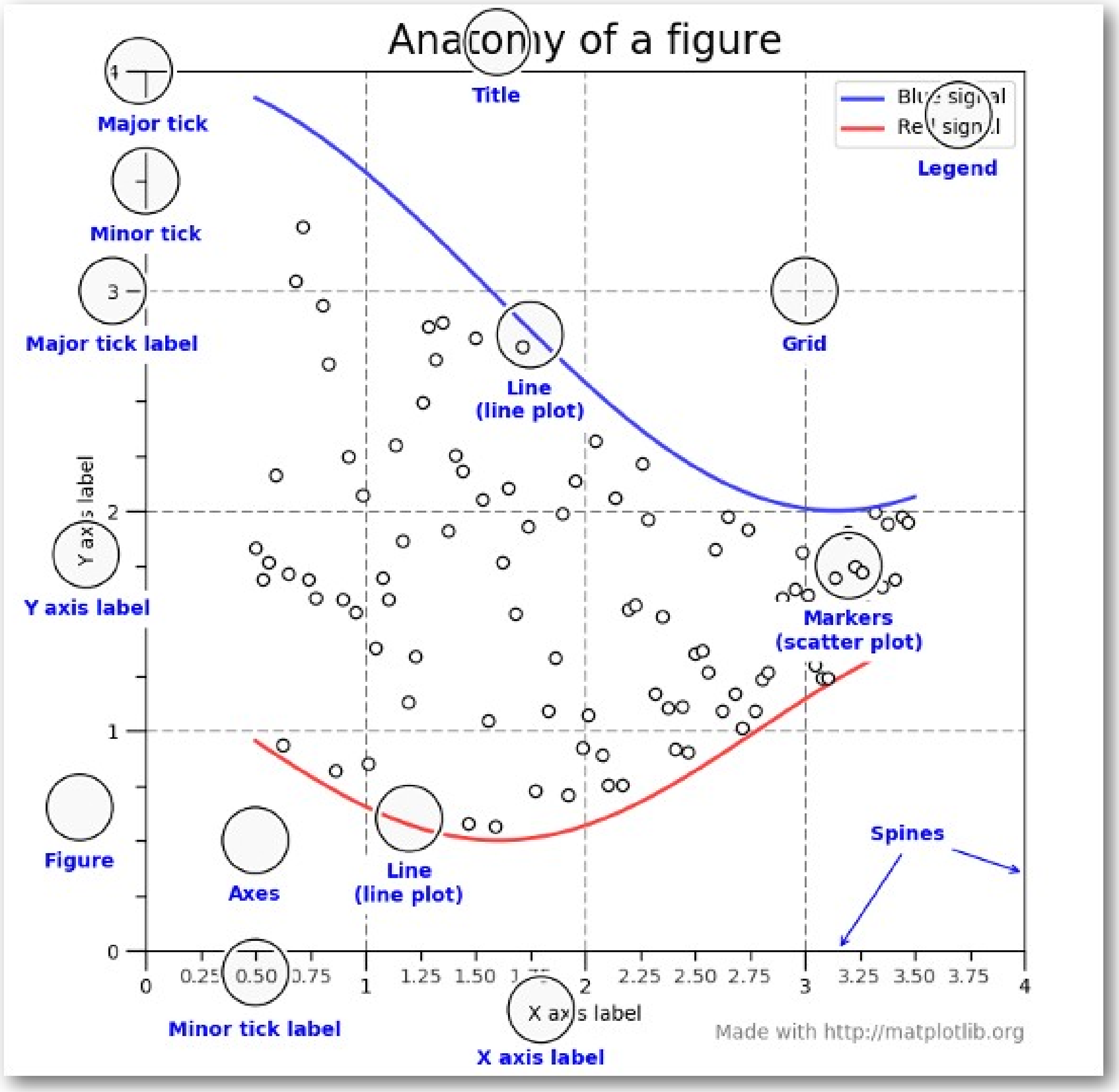
```
x4 = range(1, 5)
```

```
plt.plot(x4, [xi*1.5 for xi in x4])

plt.plot(x4, [xi*3 for xi in x4])

plt.plot(x4, [xi/3.0 for xi in x4])

plt.show()
```



13. Parts of a Plot

There are different parts of a plot. These are title, legend, grid, axis and labels etc. These are denoted in the following figure:-



14. Saving the Plot

We can save the figures in a wide variety of formats. We can save them using the **savefig()** command as follows:-

```
fig.savefig('fig1.png')
```

We can explore the contents of the file using the IPython **Image** object.

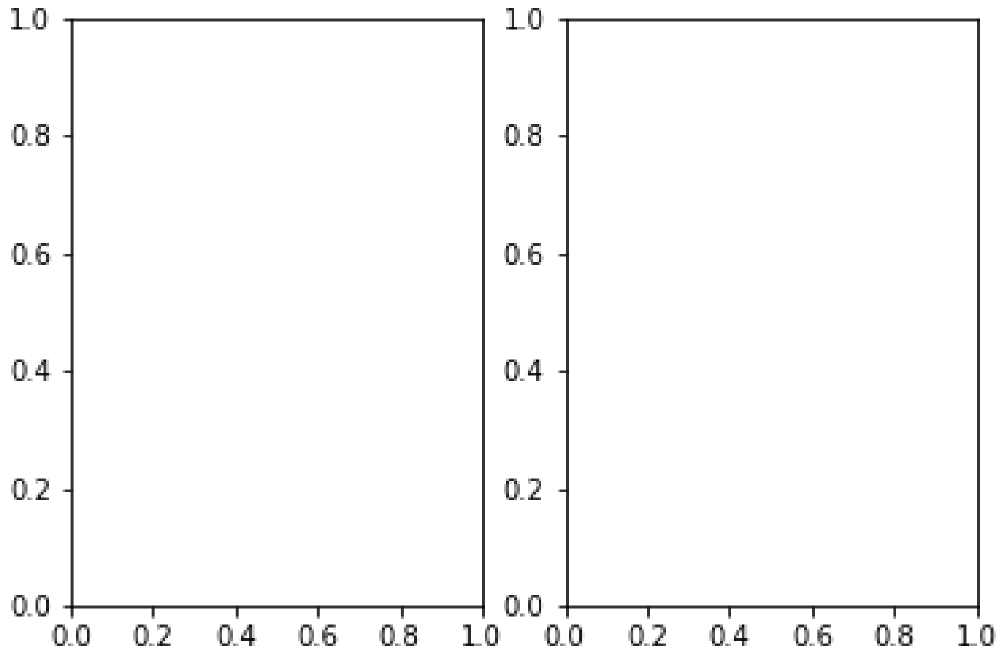
```
from IPython.display import Image
```

```
Image('fig1.png')
```

In **savefig()** command, the file format is inferred from the extension of the given filename. Depending on the backend, many different file formats are available. The list of supported file types can be found by using the get_supported_filetypes() method of the figure canvas object as follows:-

`fig.canvas.get_supported_filetypes()`

```
# Saving the figure

fig.savefig('plot1.png')

# Explore the contents of figure

from IPython.display import Image

Image('plot1.png')
```

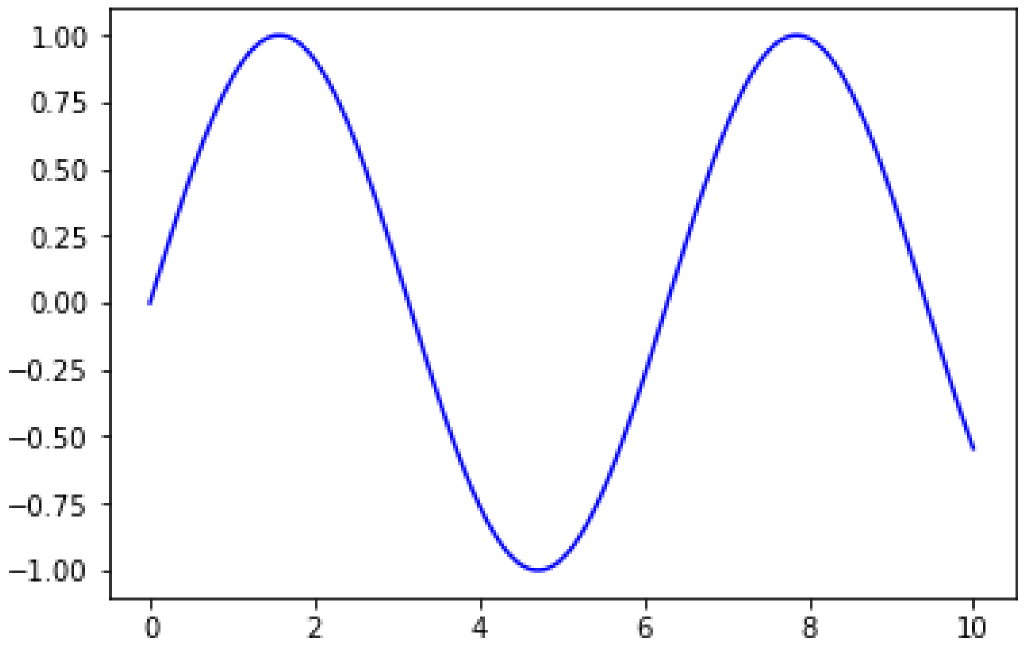


```
# Explore supported file formats


fig.canvas.get_supported_filetypes()

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

15. Line Plot

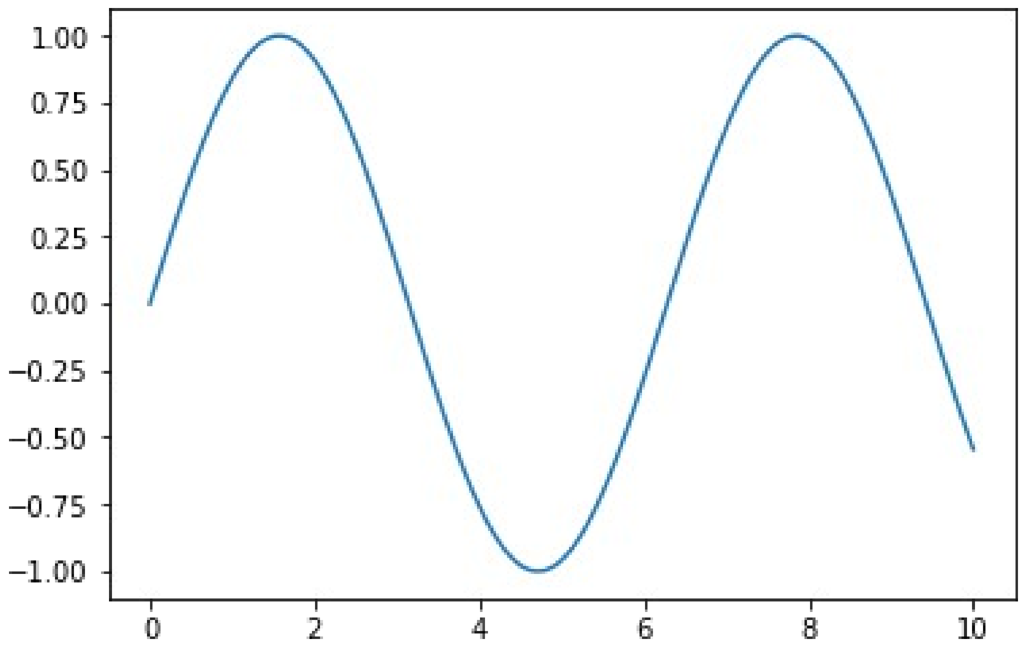
We can use the following commands to draw the simple sinusoid line plot:-

```
# Create figure and axes first
fig = plt.figure()

ax = plt.axes()

# Declare a variable x5
x5 = np.linspace(0, 10, 1000)


# Plot the sinusoid function
ax.plot(x5, np.sin(x5), 'b-');
```



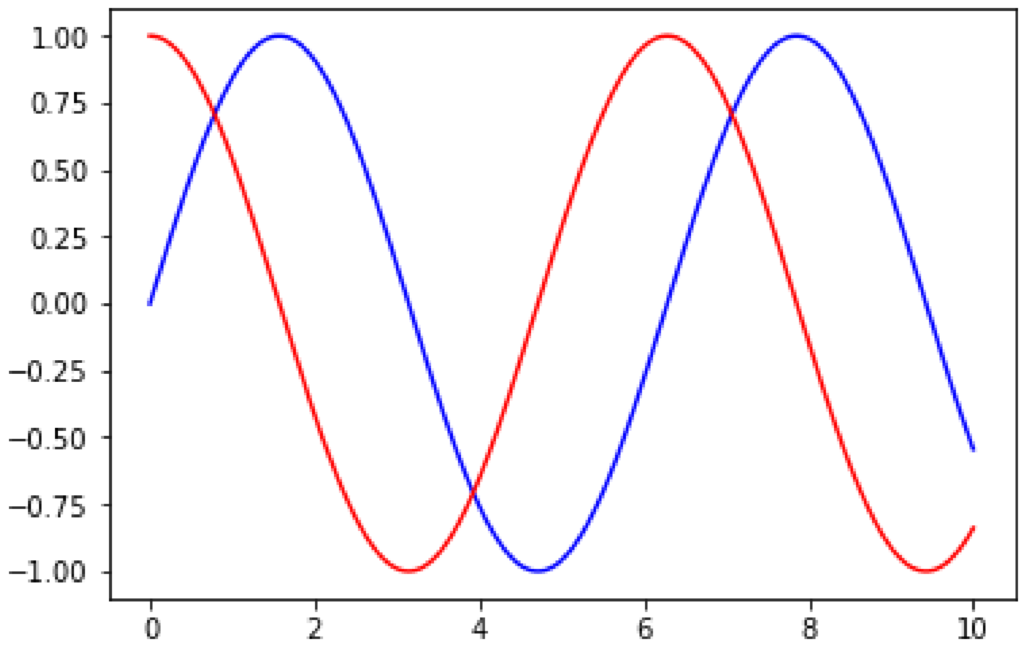
Alternatively, we can use the Pyplot interface and let the figure and axes be created for us in the background.

```
# Plot the sinusoid function

plt.plot(x5, np.sin(x5));
```



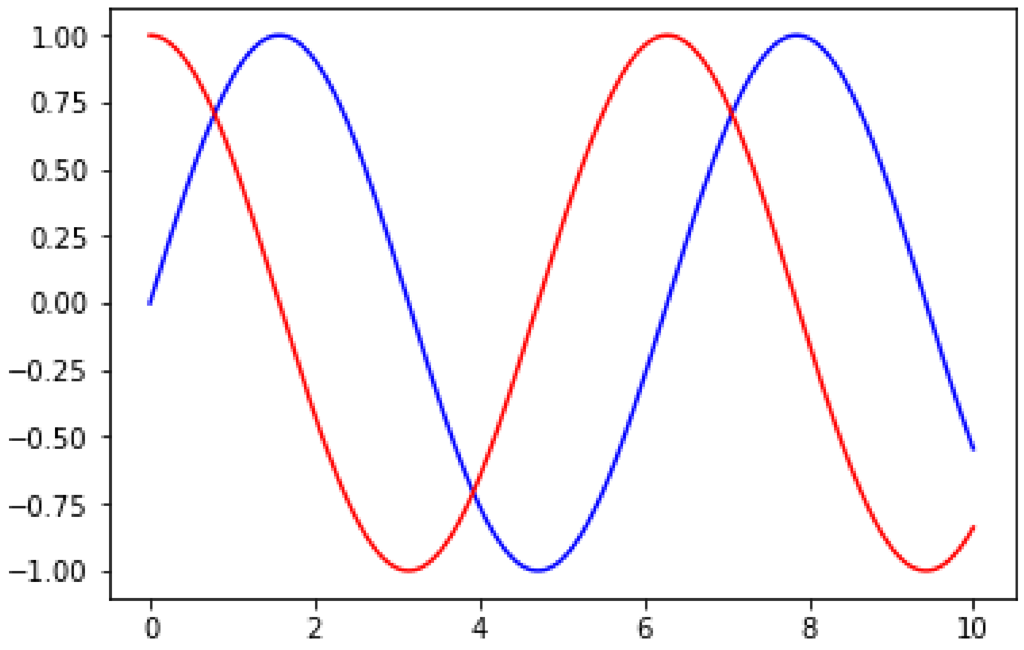
We can draw multiple lines in a single figure in OO API as follows:-

```
fig = plt.figure()

ax = plt.axes()

x6 = np.linspace(0, 10, 1000)

ax.plot(x6, np.sin(x6), 'b-')

ax.plot(x6, np.cos(x6), 'r-');
```



Similarly, if we want to create a single figure with multiple lines, we can call the plot function multiple times.

```
plt.plot(x6, np.sin(x6), 'b-')

plt.plot(x6, np.cos(x6), 'r-');
```



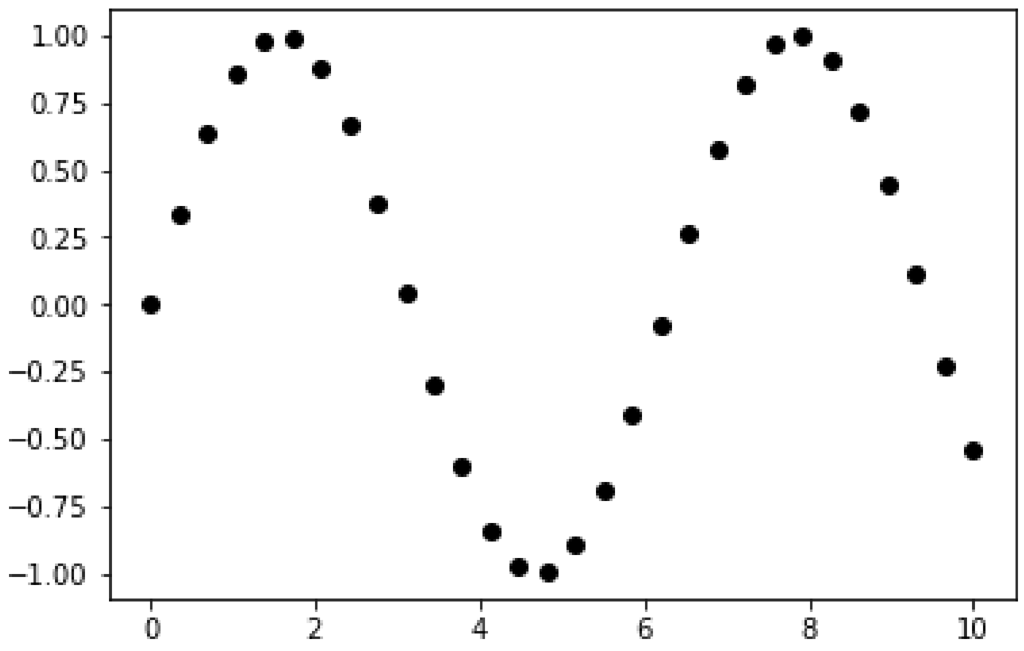
16. Scatter Plot

Another commonly used plot type is the scatter plot. Here the points are represented individually with a dot or a circle.

Scatter Plot with plt.plot()

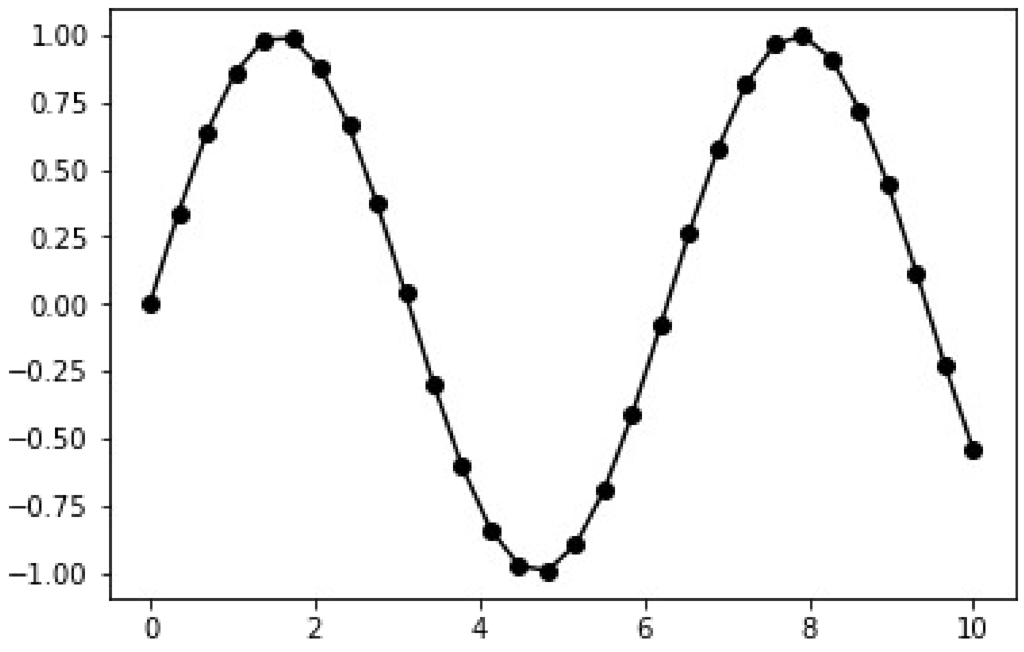
We have used plt.plot/ax.plot to produce line plots. We can use the same functions to produce the scatter plots as follows:-

```
x7 = np.linspace(0, 10, 30)

y7 = np.sin(x7)

plt.plot(x7, y7, 'o', color = 'black');
```



We can customize the plot by combining the character codes together with line and color codes to plot points along with a line connecting them as follows:-

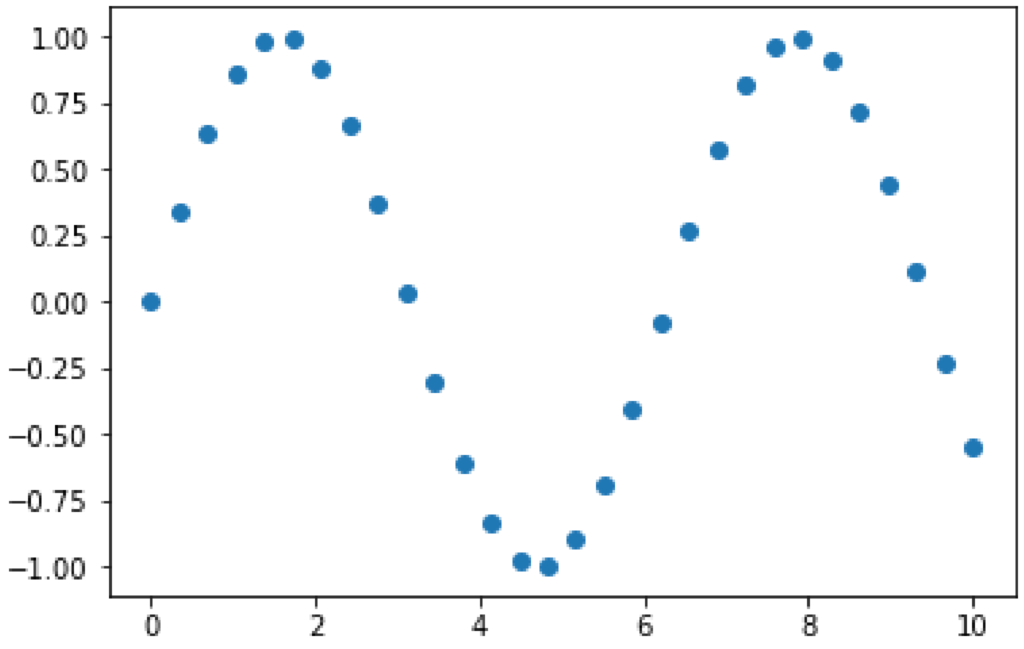
```
plt.plot(x7, y7, '-ok');
```



Scatter Plot with plt.scatter()

An alternative approach of creating scatter plot is to use plt.scatter() function. It is more powerful method of creating scatter plots than to use the plt.plot() function. We can use plt.scatter() function as follows:-

```
plt.scatter(x7, y7, marker='o')
plt.show()
```



plt.plot() Vs plt.scatter() functions

For smaller datasets, it does not matter whether we use plt.plot() or plt.scatter() functions to create scatter-plots.

But for larger datasets, plt.plot() function is more efficient than plt.scatter() function. In plt.plot() function, the points are clones of each other. So the work of determining the appearance of the points is done only once for the entire set of data. In plt.scatter() function, the renderer must do the extra work of constructing each point individually.

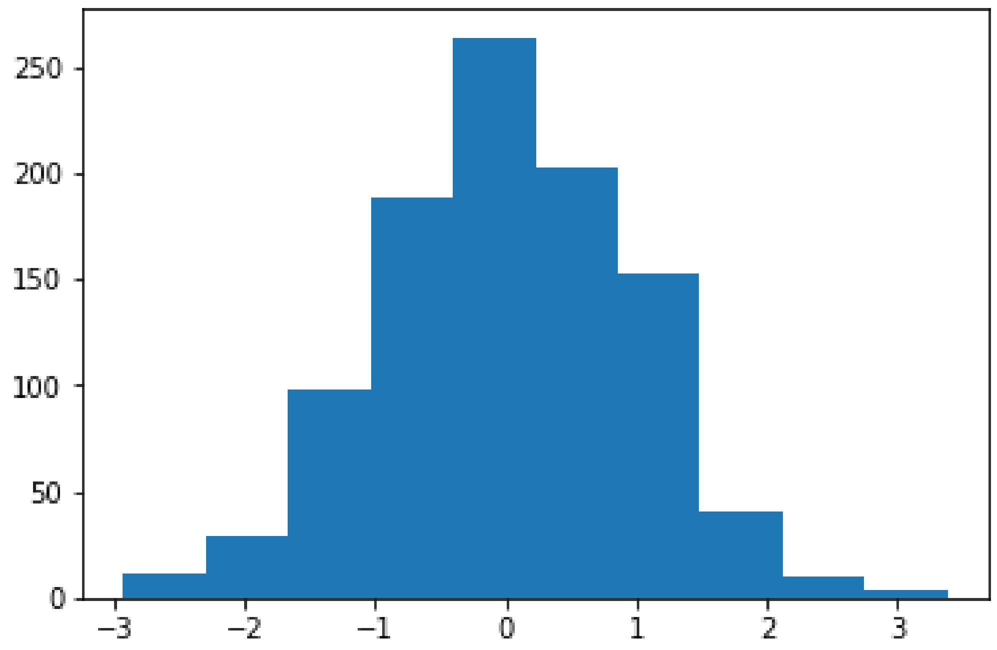
So, for large datasets, the difference between plt.plot() and plt.scatter() functions can lead to vastly different performance. Hence, plt.plot() function should be preferred over plt.scatter() function for large datasets.

17. Histogram

Histogram charts are a graphical display of frequencies. They are represented as bars. They show what portion of the dataset falls into each category, usually specified as non-overlapping intervals. These categories are called bins.

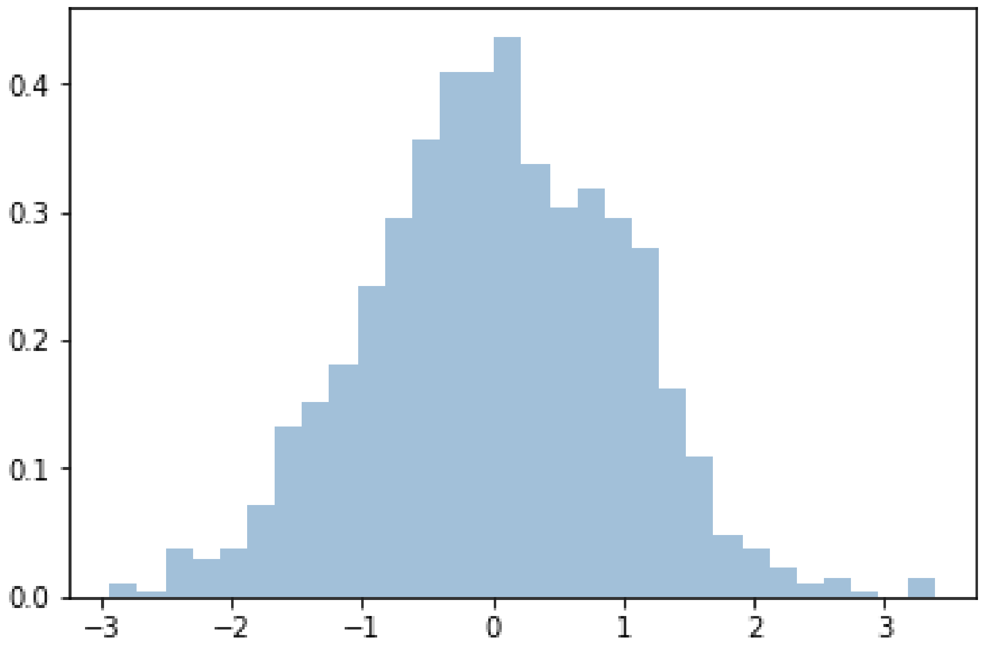
The **plt.hist()** function can be used to plot a simple histogram as follows:-

```
data1 = np.random.randn(1000)

plt.hist(data1);
```



The **plt.hist()** function has many parameters to tune both the calculation and the plot display. We can see these parameters in action as follows:-

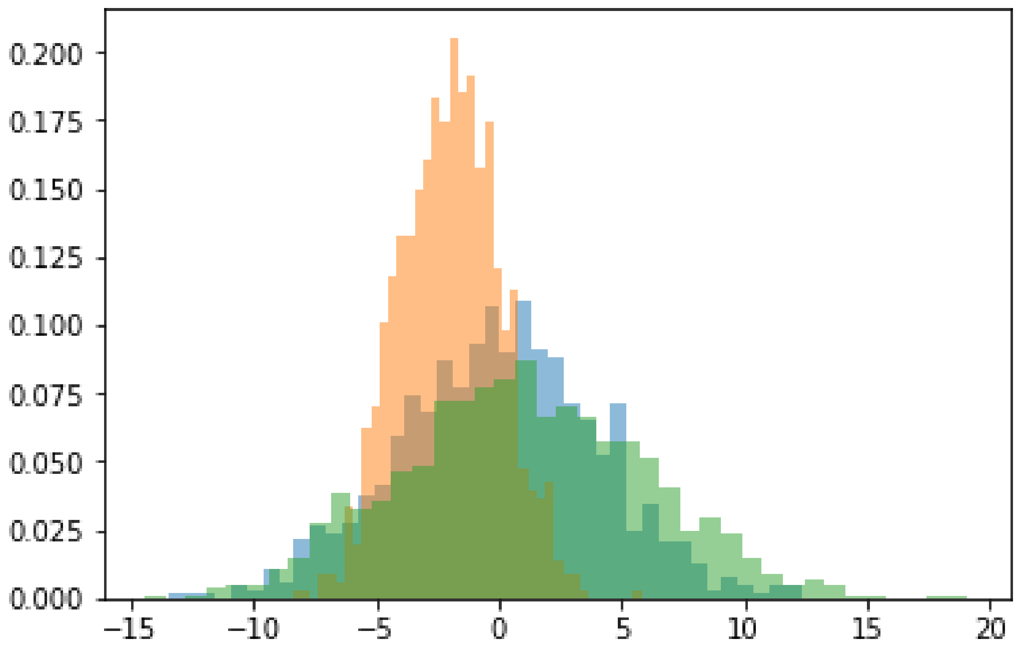
```
plt.hist(data1, bins=30, density=True, alpha=0.5,
histtype='stepfilled', color='steelblue')
plt.show();
```



Compare Histograms of several distributions

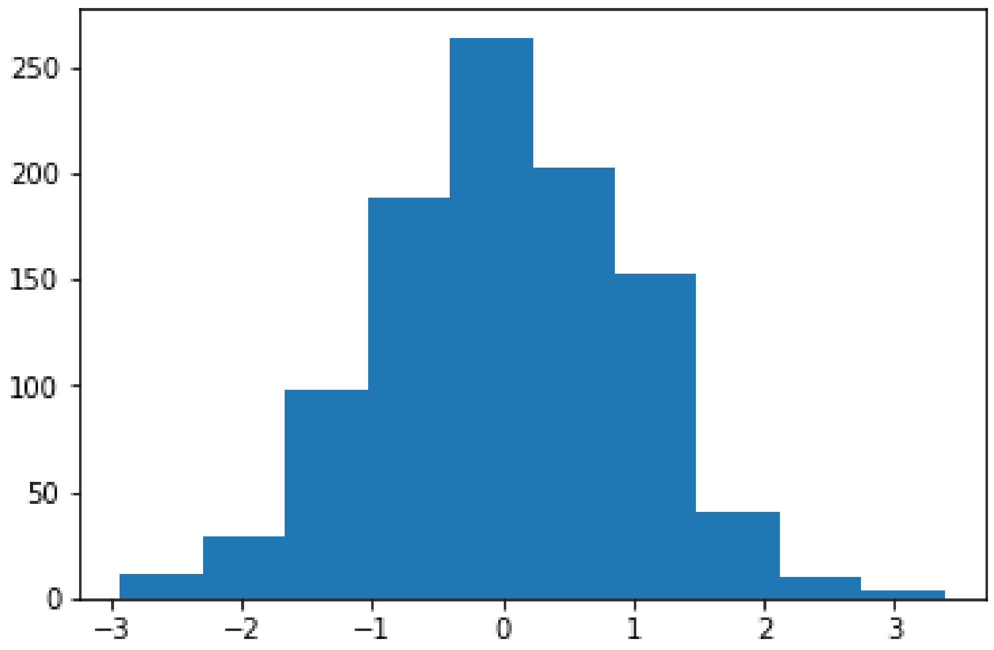
We can use the combination of histtype='stepfilled' along with transparency level alpha to compare histograms of several distributions.

```
x1 = np.random.normal(0, 4, 1000)
x2 = np.random.normal(-2, 2, 1000)
x3 = np.random.normal(1, 5, 1000)

kwargs = dict(histtype='stepfilled', alpha = 0.5, density = True, bins
= 40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs)

plt.show();
```



Create Histogram with OO API

```
# Plots in Matplotlib reside within a figure object - Use plt.figure()
to create new figure object
fig = plt.figure()

# Create subplots using add_subplot() function
ax = fig.add_subplot(1, 1, 1)

# Plot histogram
ax.hist(data1, bins=10)
plt.show();
```

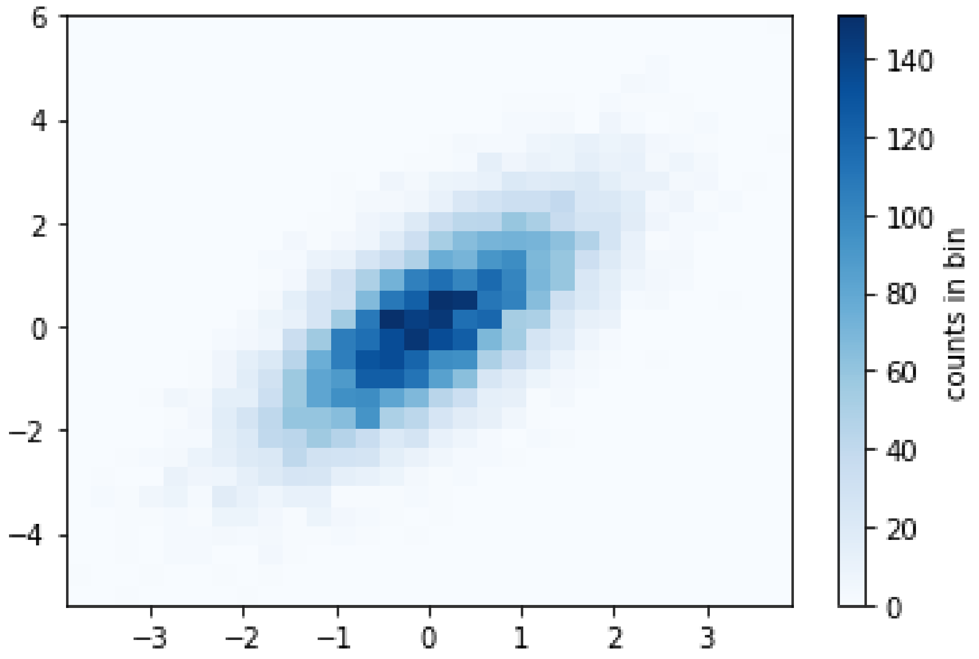


Two-Dimensional Histograms

Just as we create histograms in one dimension, we can also create histograms in two-dimensions by dividing points among two-dimensional bins.

We can use Matplotlib's plt.hist2d() function to plot a two-dimensional histogram as follows:-

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x8, y8 = np.random.multivariate_normal(mean, cov, 10000).T

plt.hist2d(x8, y8, bins = 30, cmap = 'Blues')

cb = plt.colorbar()

cb.set_label('counts in bin')
```

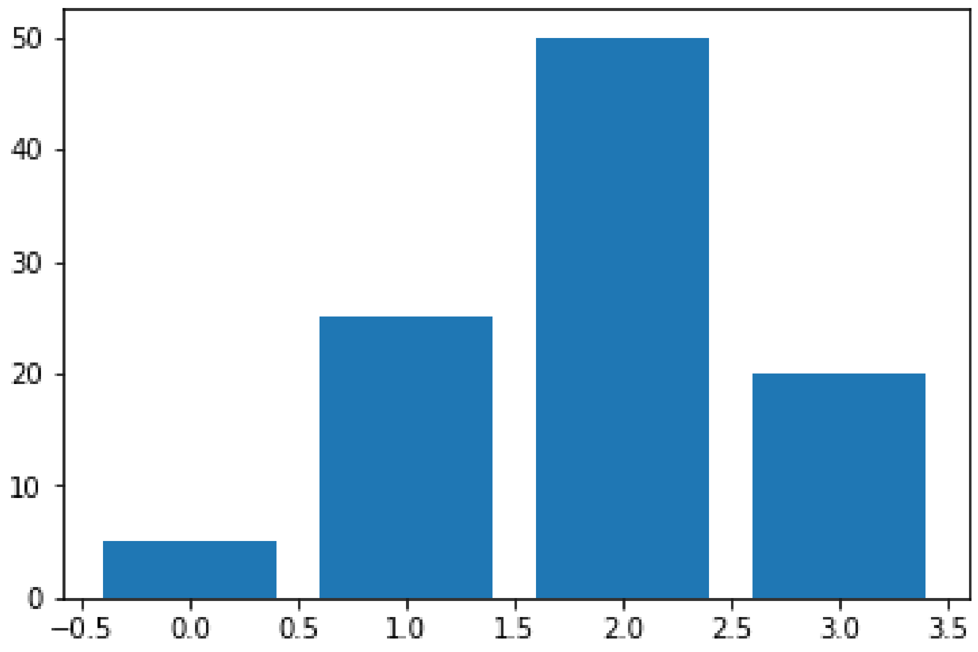


18. Bar Chart

Bar charts display rectangular bars either in vertical or horizontal form. Their length is proportional to the values they represent. They are used to compare two or more values.

We can plot a bar chart using plt.bar() function. We can plot a bar chart as follows:-

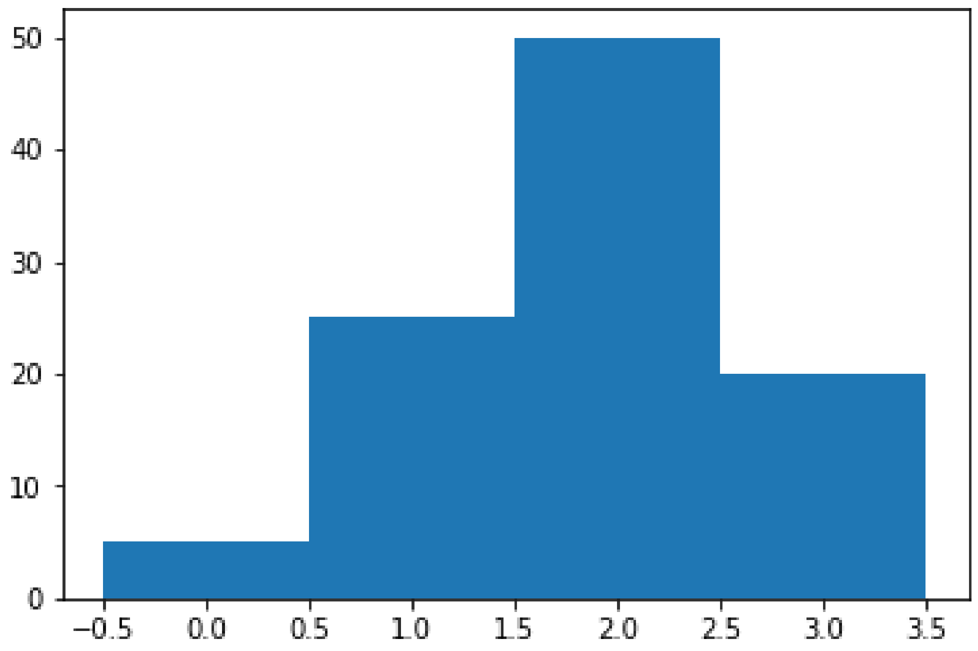
```
data2 = [5. , 25. , 50. , 20.]

plt.bar(range(len(data2)), data2)

plt.show()
```



The thickness of the Bar Chart

By default, a bar will have a thickness of 0.8 units. If we draw a bar of unit length, we have a gap of 0.2 between them. We can change the thickness of the bar chart by setting width parameter to 1.

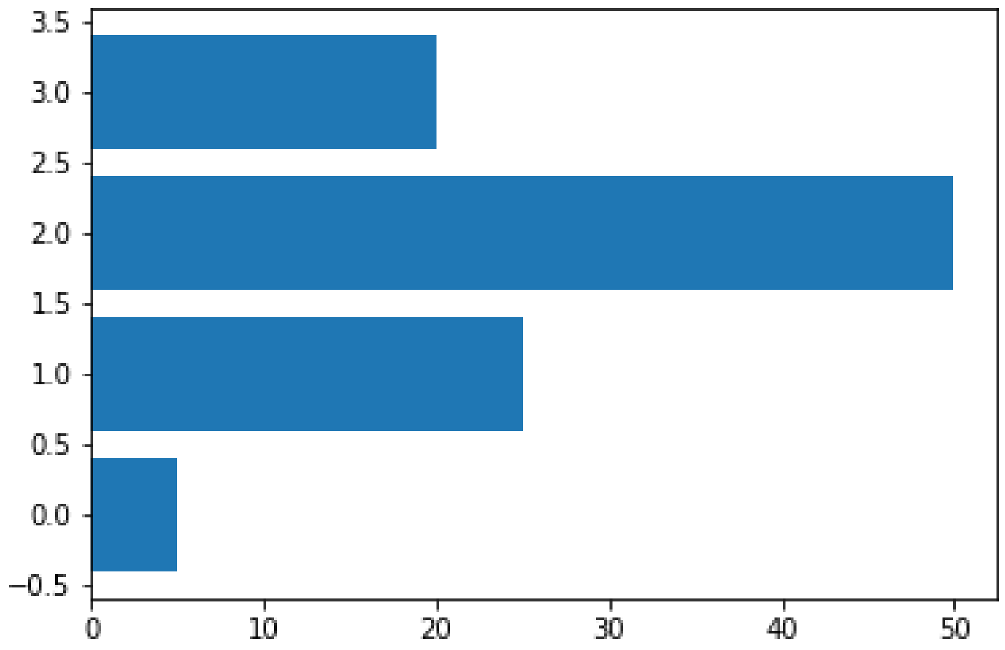
```
plt.bar(range(len(data2)), data2, width = 1)

plt.show()
```



19. Horizontal Bar Chart

We can produce Horizontal Bar Chart using the plt.barh() function. It is the strict equivalent of plt.bar() function.

```
data2 = [5. , 25. , 50. , 20.]

plt.barh(range(len(data2)), data2)

plt.show()
```

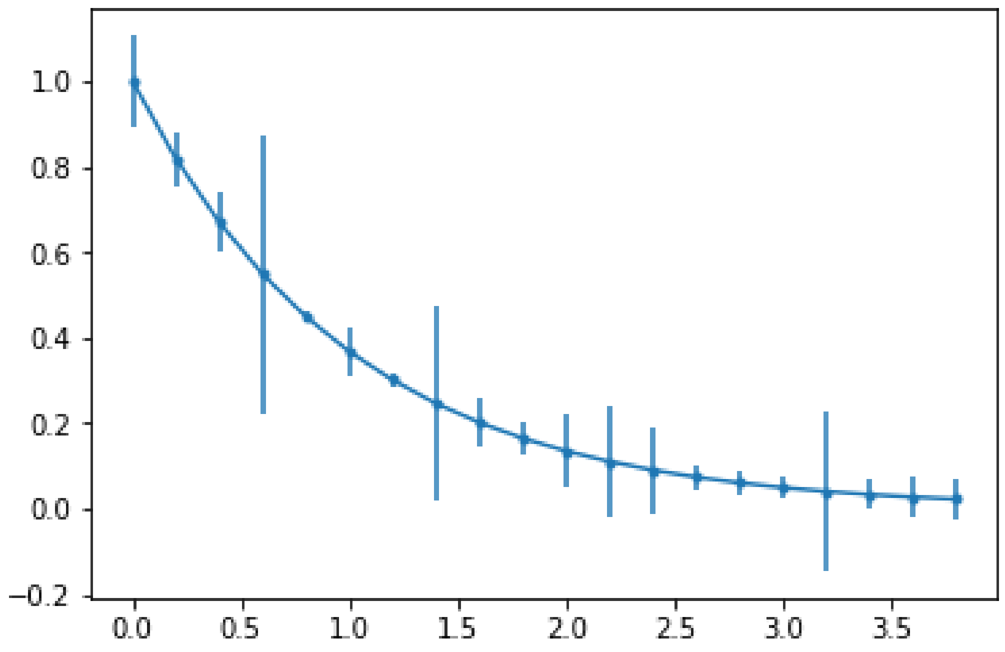


20. Error Bar Chart

In experimental design, the measurements lack perfect precision. So, we have to repeat the measurements. It results in obtaining a set of values. The representation of the distribution of data values is done by plotting a single data point (known as mean value of dataset) and an error bar to represent the overall distribution of data.

We can use Matplotlib's **errorbar()** function to represent the distribution of data values. It can be done as follows:-

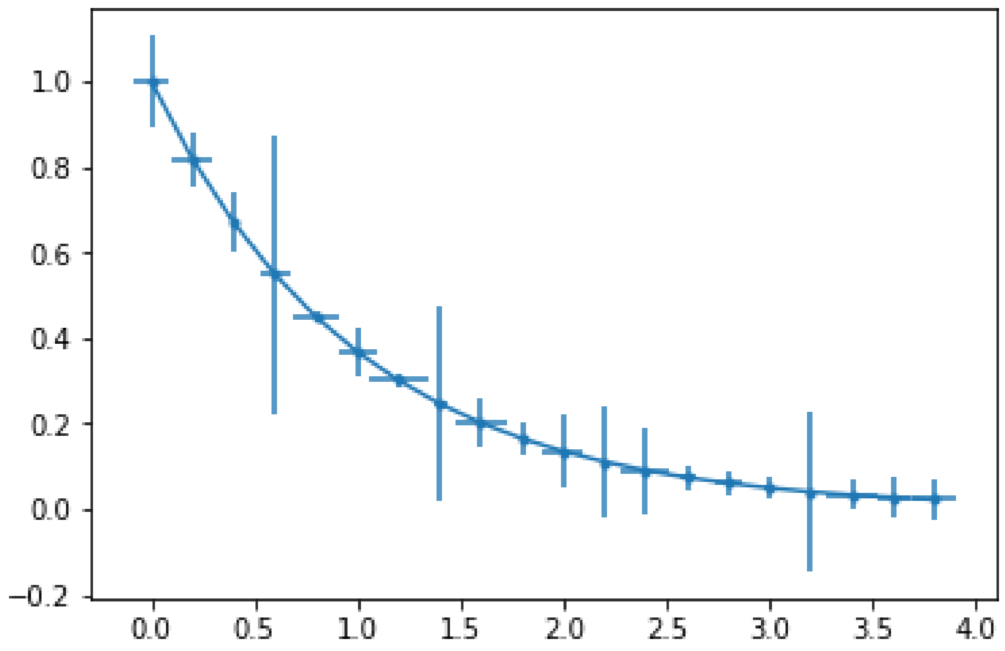
```
x9 = np.arange(0, 4, 0.2)

y9 = np.exp(-x9)

e1 = 0.1 * np.abs(np.random.randn(len(y9)))

plt.errorbar(x9, y9, yerr = e1, fmt = '.-')

plt.show();
```



I have plotted x versus y with error deltas as vertical error bars, as specified by the **yerr** keyword argument. There is an equivalent argument, **xerr**, to draw horizontal error bars.

We can also specify both xerr and yerr together as follows:-

```
e2 = 0.1 * np.abs(np.random.randn(len(y9)))

plt.errorbar(x9, y9, yerr=e1, xerr=e2, fmt = '.-', capsize=0)

plt.show();
```



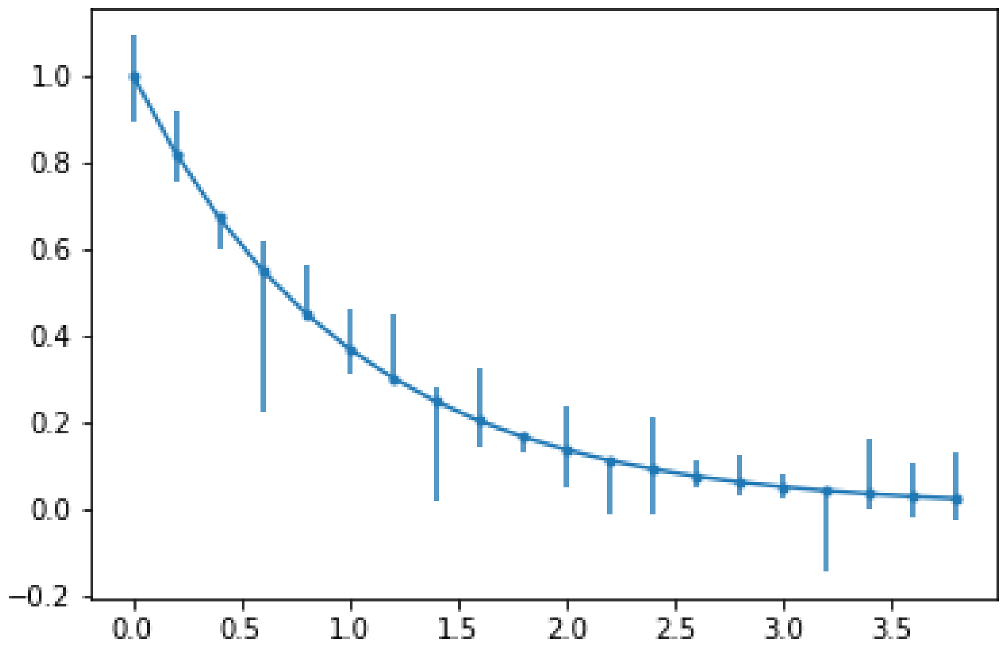
We can see that, for each point we have four different errors. One for each direction: -x, +x, -y and +y.

Asymmetrical Error Bars

The error bars described in the previous section, are called **symmetrical error bars**. Their negative error is equal in value to the positive error. So error bar is symmetrical to the point where it is drawn.

There is another type of error bar, which is **asymmetrical error bar**. To draw **asymmetrical error bars**, we have to pass two lists(or a 2D array) of values to yerr and/or xerr - the first list is for negative errors while the second list is for positive errors.

```
plt.errorbar(x9, y9, yerr=[e1, e2], fmt='.-')

plt.show();
```

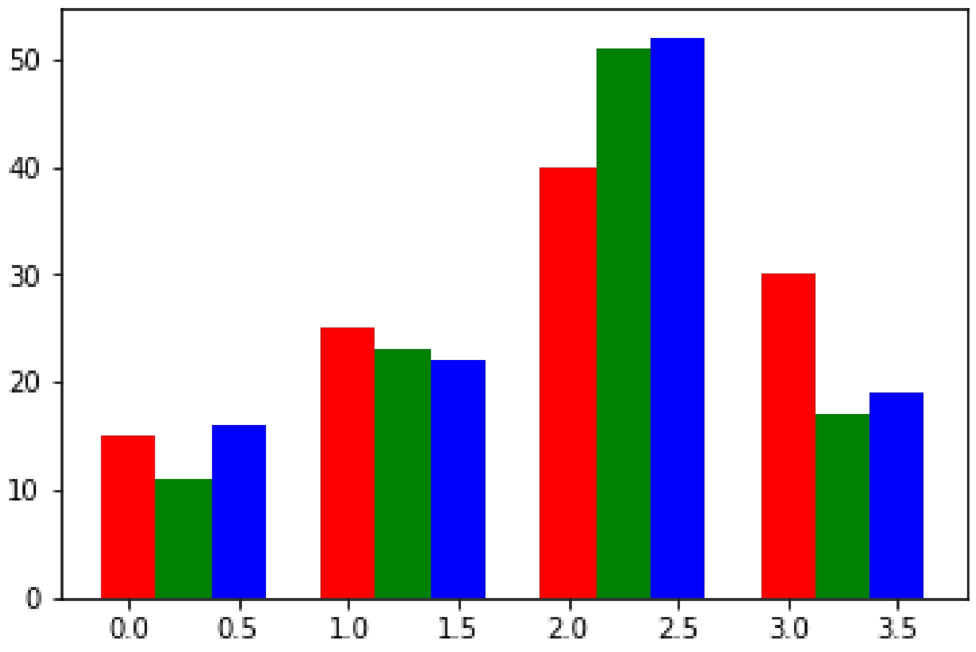


21. Multiple Bar Chart

When we want to compare several quantities while changing one variable, we might want to plot a Multiple Bar Chart. A Multiple Bar Chart is a bar chart where each variable is reflected by bars of different colours.

It can be plotted by adjusting the thickness and positions of the bars as follows:-

```
data3 = [[15., 25., 40., 30.],
         [11., 23., 51., 17.],
         [16., 22., 52., 19.]]

z1 = np.arange(4)

plt.bar(z1 + 0.00, data3[0], color = 'r', width = 0.25)
plt.bar(z1 + 0.25, data3[1], color = 'g', width = 0.25)
plt.bar(z1 + 0.50, data3[2], color = 'b', width = 0.25)

plt.show()
```

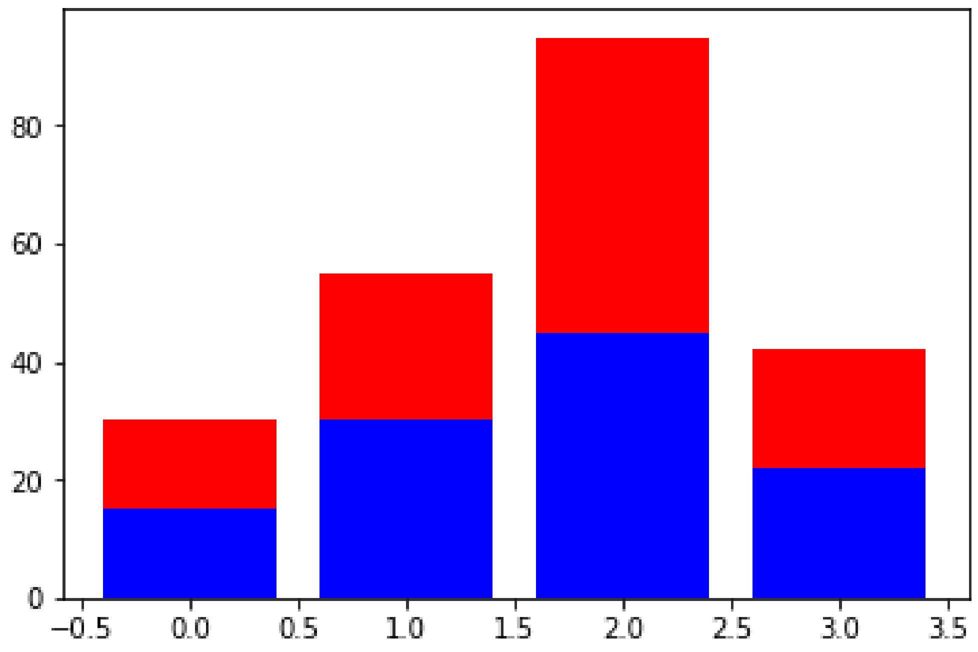


The data3 variable contains three series of four values. The code snippet shows three bar charts of four bars. The bars will have a thickness of 0.25 units. Each bar chart will be shifted 0.25 units from the previous one. Color has been added for clear presentation and understanding.

22. Stacked Bar Chart

We can draw stacked bar chart by using a special parameter called **bottom** from the plt.bar() function. It can be done as follows:-

```
A = [15., 30., 45., 22.]

B = [15., 25., 50., 20.]

z2 = range(4)

plt.bar(z2, A, color = 'b')
plt.bar(z2, B, color = 'r', bottom = A)

plt.show()
```

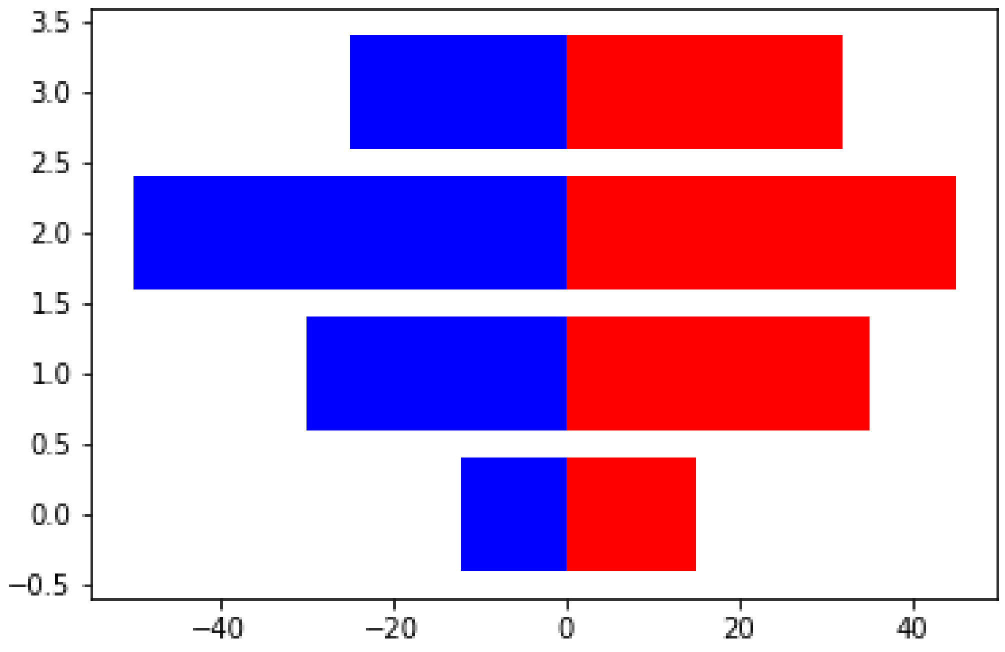


The optional **bottom** parameter of the plt.bar() function allows us to specify a starting position for a bar. Instead of running from zero to a value, it will go from the bottom to value. The first call to plt.bar() plots the blue bars. The second call to plt.bar() plots the red bars, with the bottom of the red bars being at the top of the blue bars.

23. Back-to-Back Bar Charts

We can plot two bar charts back to back at the same time using a simple trick. The idea is to have two bar charts, using a simple trick, that is, the length/height of one bar can be negative.

```
U1 = np.array([15., 35., 45., 32.])
U2 = np.array([12., 30., 50., 25.])

z1 = np.arange(4)

plt.barh(z1, U1, color = 'r')
plt.barh(z1, -U2, color = 'b')

plt.show()
```

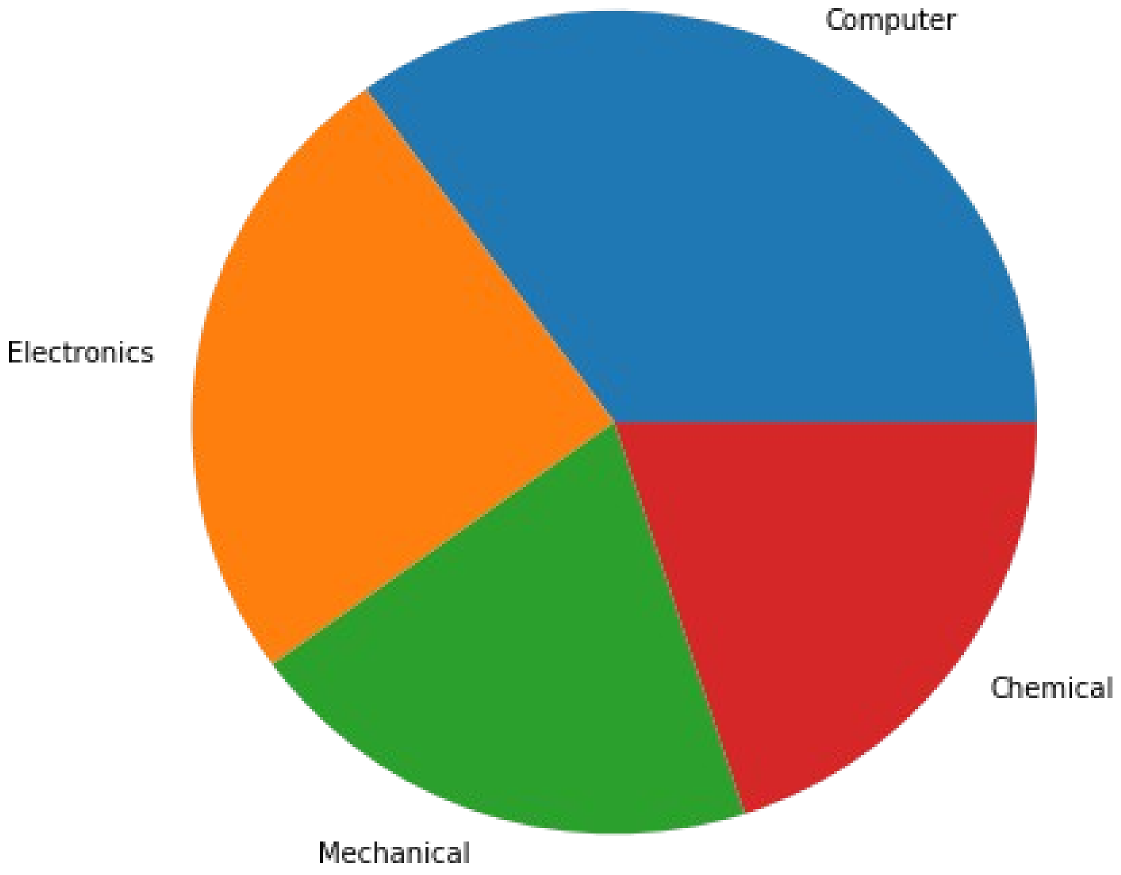


24. Pie Chart

Pie charts are circular representations, divided into sectors. The sectors are also called **wedges**. The arc length of each sector is proportional to the quantity we are describing. It is an effective way to represent information when we are interested mainly in comparing the wedge against the whole pie, instead of wedges against each other.

Matplotlib provides the **pie()** function to plot pie charts from an array X. Wedges are created proportionally, so that each value x of array X generates a wedge proportional to x/sum(X).

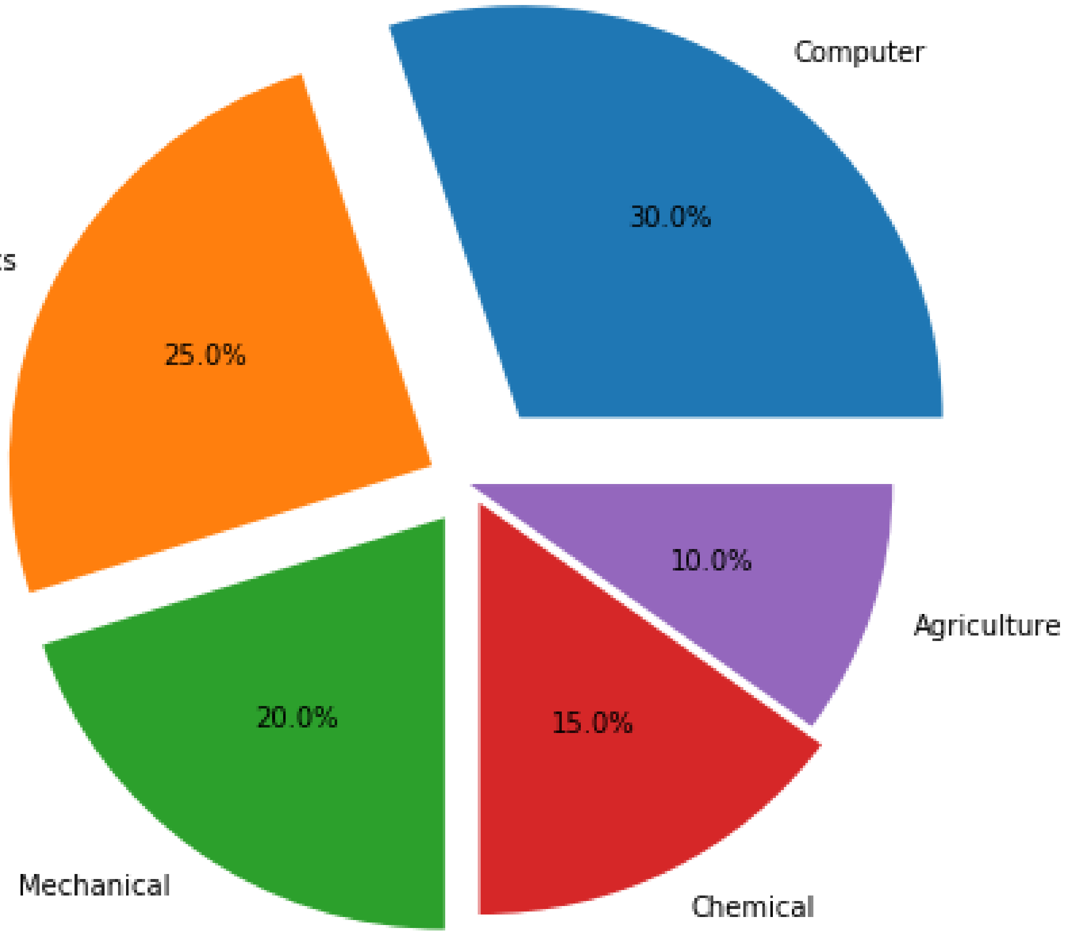
```
plt.figure(figsize=(7,7))

x10 = [35, 25, 20, 20]

labels = ['Computer', 'Electronics', 'Mechanical', 'Chemical']

plt.pie(x10, labels=labels);

plt.show()
```



Exploded Pie Chart

We can plot an exploded Pie chart with the addition of keyword argument **explode**. It is an array of the same length as that of X. Each of its values specify the radius fraction with which to offset the wedge from the center of the pie.

```
plt.figure(figsize=(7,7))

x11 = [30, 25, 20, 15, 10]

labels = ['Computer', 'Electronics', 'Mechanical', 'Chemical',
'Agriculture']

explode = [0.2, 0.1, 0.1, 0.05, 0]

plt.pie(x11, labels=labels, explode=explode, autopct='%1.1f%%');
plt.show()
```

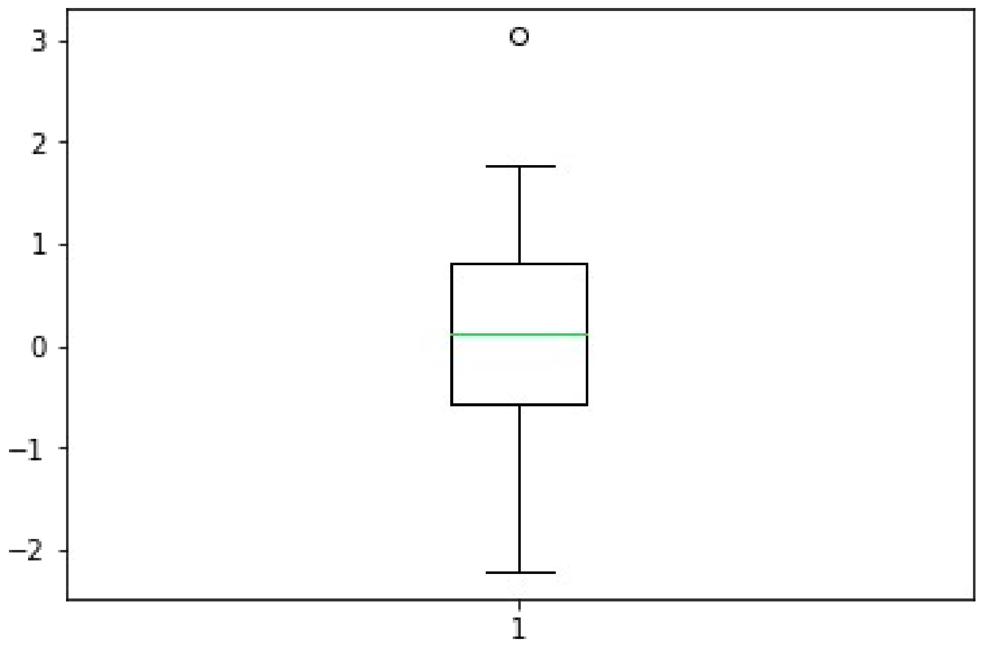


25. Boxplot

Boxplot allows us to compare distributions of values by showing the median, quartiles, maximum and minimum of a set of values.

We can plot a boxplot with the **boxplot()** function as follows:-

```
data3 = np.random.randn(100)

plt.boxplot(data3)

plt.show();
```



The **boxplot()** function takes a set of values and computes the mean, median and other statistical quantities. The following points describe the preceeding boxplot:

• The red bar is the median of the distribution.

• The blue box includes 50 percent of the data from the lower quartile to the upper quartile. Thus, the box is centered on the median of the data.

• The lower whisker extends to the lowest value within 1.5 IQR from the lower quartile.

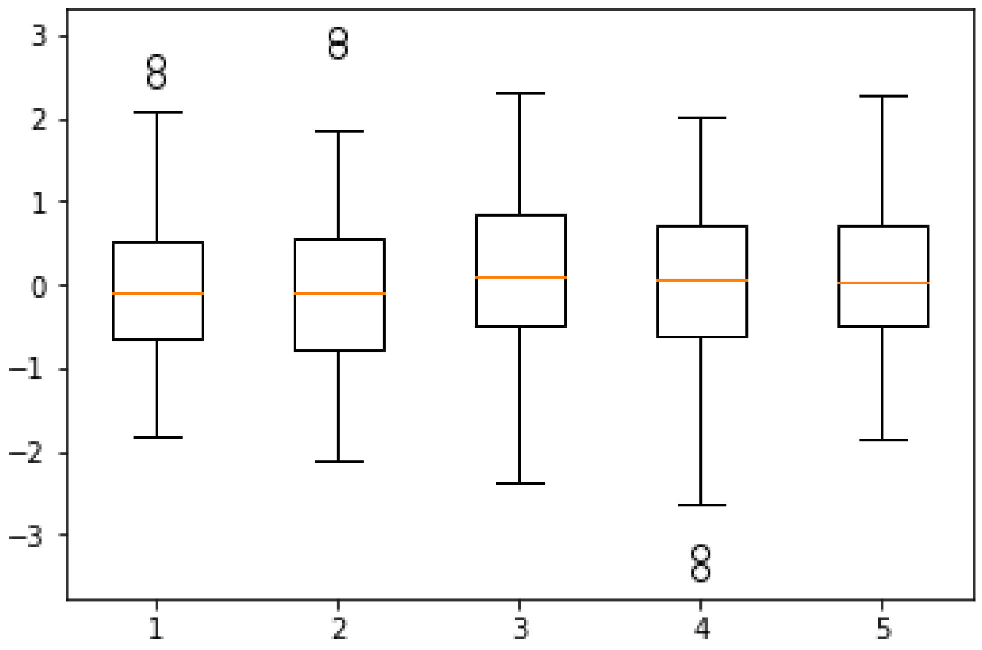
• The upper whisker extends to the highest value within 1.5 IQR from the upper quartile.

• Values further from the whiskers are shown with a cross marker.

Customized Boxplot

To show more than one boxplot in a single graph, calling plt.boxplot() once for each boxplot is not going to work. It will simply draw the boxplots over each other. We can draw several boxplots with just one single call to plt.boxplot() as follows:

```
data4 = np.random.randn(100, 5)

plt.boxplot(data4)

plt.show();
```

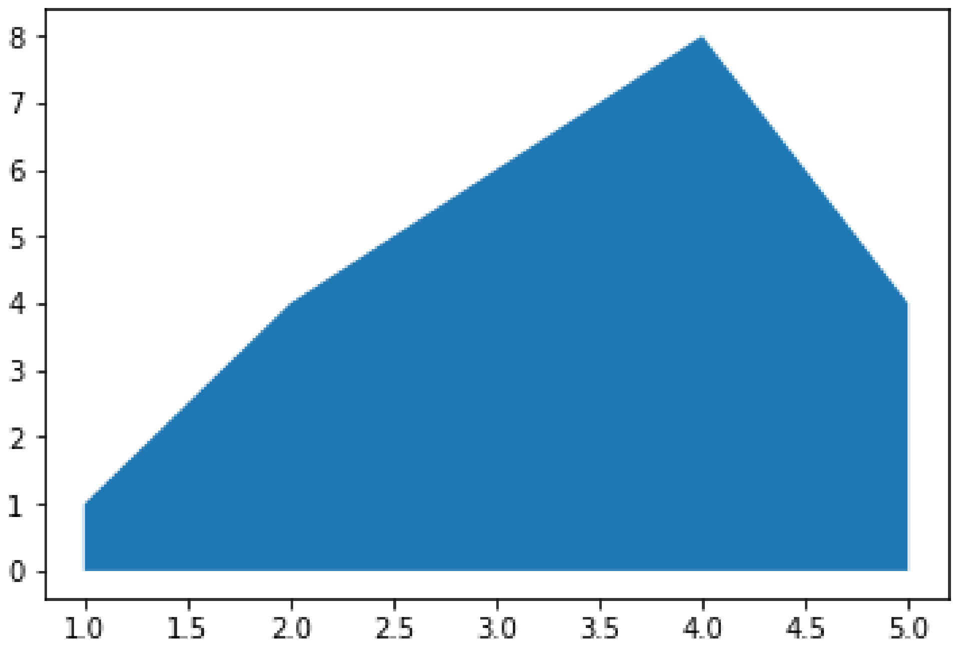


26. Area Chart

An **Area Chart** is very similar to a **Line Chart**. The area between the x-axis and the line is filled in with color or shading. It represents the evolution of a numerical variable following another numerical variable.

We can create an Area Chart as follows:-

```
# Create some data
x12 = range(1, 6)
y12 = [1, 4, 6, 8, 4]

# Area plot
plt.fill_between(x12, y12)
plt.show()
```



I have created a basic Area chart. I could also use the stackplot function to create the Area chart as follows:-

`plt.stackplot(x12, y12)`

The fill_between() function is more convenient for future customization.

27. Contour Plot

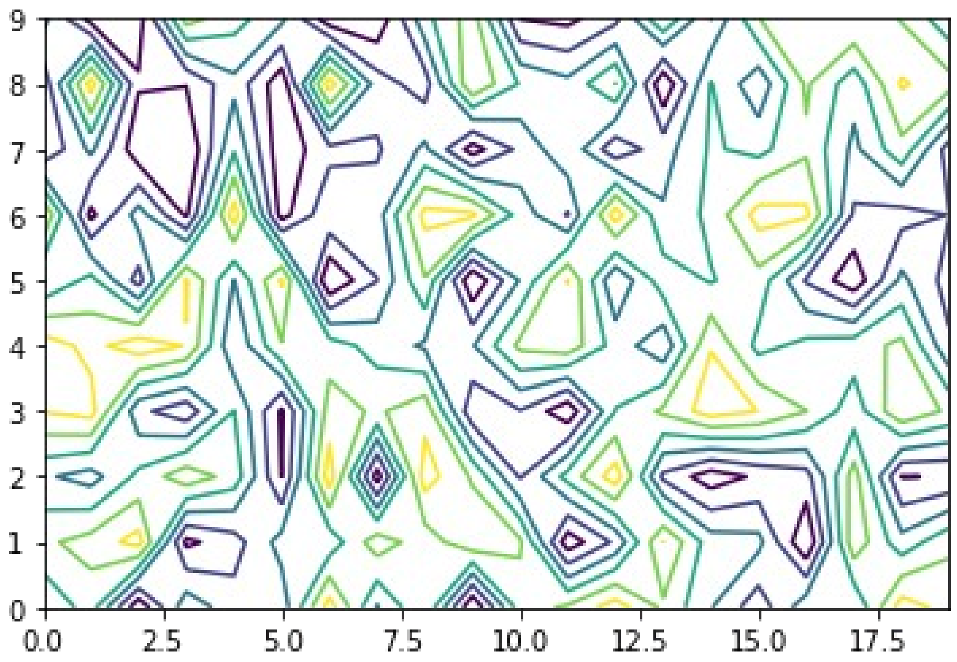
Contour plots are useful to display three-dimensional data in two dimensions using contours or color-coded regions. **Contour lines** are also known as **level lines** or **isolines**. **Contour lines** for a function of two variables are curves where the function has constant values. They have specific names beginning with iso- according to the nature of the variables being mapped.

There are lot of applications of **Contour lines** in several fields such as meteorology(for temperature, pressure, rain, wind speed), geography, magnetism, engineering, social sciences and so on.

The density of the lines indicates the **slope** of the function. The **gradient** of the function is always perpendicular to the contour lines. When the lines are close together, the length of the gradient is large and the variation is steep.

A **Contour plot** can be created with the **plt.contour()** function as follows:-

```
# Create a matrix
matrix1 = np.random.rand(10, 20)

cp = plt.contour(matrix1)

plt.show()
```



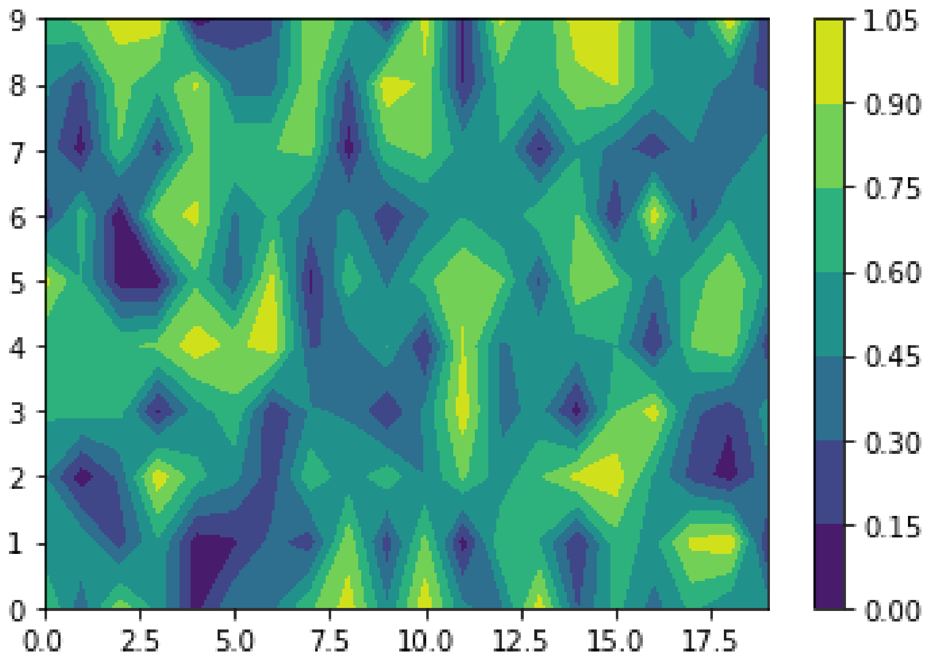
The **contour()** function draws contour lines. It takes a 2D array as input.Here, it is a matrix of 10 x 20 random elements.

The number of level lines to draw is chosen automatically, but we can also specify it as an additional parameter, N.

`plt.contour(matrix, N)`

There is also a similar function that draws a filled contours plot, **contourf()**. We can use this function as follows:-

```
csf = plt.contourf(matrix1)

plt.colorbar()

plt.show()
```

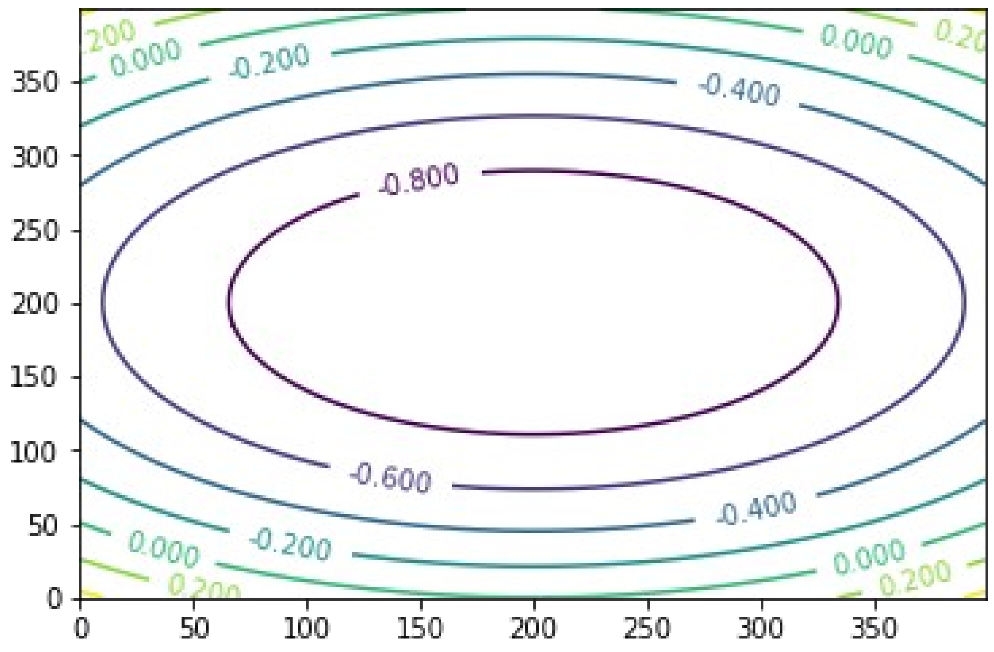


contourf() fills the spaces between the contours lines with the same color progression used in the **contour()** plot. Contour colors can be changed using a **colormap**. A **colormap** is a set of colors used as a lookup table by Matplotlib when it needs to select more colors. I also added a **colorbar()** call to draw a color bar next to the plot to identify the ranges the colors are assigned to.

Labeling the level lines is important in order to provide information about what levels were chosen to display. **clabel()** does this by taking as input a contour instance.

I draw several ellipses and then call **clabel()** to display the selected levels. The output of the code is shown below:-

```
x13 = np.arange(-2, 2, 0.01)
y13 = np.arange(-2, 2, 0.01)

X, Y = np.meshgrid(x13, y13)

ellipses = X*X/9 + Y*Y/4 - 1

cs = plt.contour(ellipses)

plt.clabel(cs)

plt.show()
```



28. Image Plot

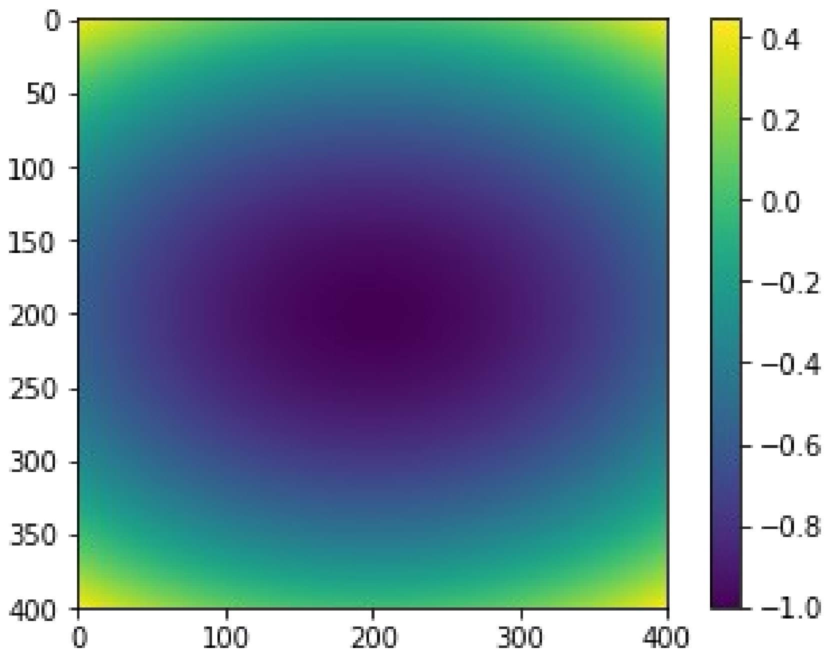
Matplotlib has basic image plotting capabilities provided by the functions: **imread()** and **imshow()**.

imread() reads an image from a file and converts it into a NumPy array. Once the image is an array, we can do all the transformations we like.

imshow() takes an array as input and displays it on the screen. **imshow()** can plot any 2D sets of data and not just the ones read from image files.

The output of **imshow()** is shown below:-

```
x13 = np.arange(-2, 2, 0.01)
y13 = np.arange(-2, 2, 0.01)

X, Y = np.meshgrid(x13, y13)

ellipses = X*X/9 + Y*Y/4 - 1

plt.imshow(ellipses);

plt.colorbar();

plt.show()
```



29. Polar Chart

Polar plots use a completely different coordinate system.

For all the previous images, we used the Cartesian system where we have two perpendicular lines (X-axis and Y-axis) meet at a point (the origin of axes) with precise axes directions to determine positive and negative values on both, the X and Y axes.

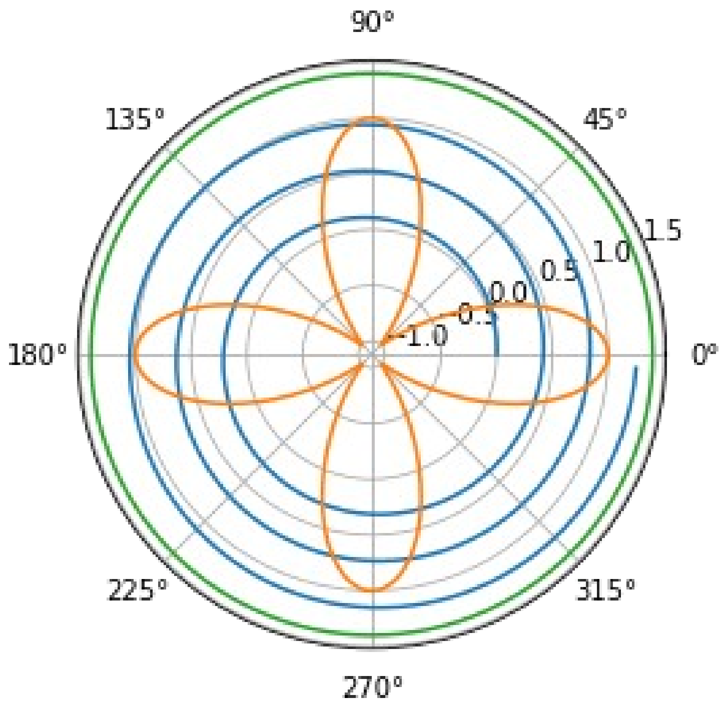
A **polar system** is a two-dimensional coordinate system, where the position of a point is expressed in terms of a radius and an angle. This system is used where the relationship between two points is better expressed using those information.

So, there are two coordinates - the **radial** and the **angular** coordinates. The radial coordinate, represented as r, denotes the point distance from the central point, called **pole**. The angular coordinate, represented as **theta**, denotes the angle required to reach the point from the polar axis.

The **polar()** Matplotlib function plots polar charts. It has two parameters which are lists of same length - **theta** for the angular coordinates and r for the radial coordinates. It is the corresponding function of **plot()** for polar charts. So, it can take multiple **theta** and **r** along with the formatting strings.

We can plot a polar chart as follows:-

```
theta = np.arange(0., 2., 1./180.)*np.pi

plt.polar(3*theta, theta/5);

plt.polar(theta, np.cos(4*theta));
plt.polar(theta, [1.4]*len(theta));

plt.show()
```



30. 3D Plotting with Matplotlib

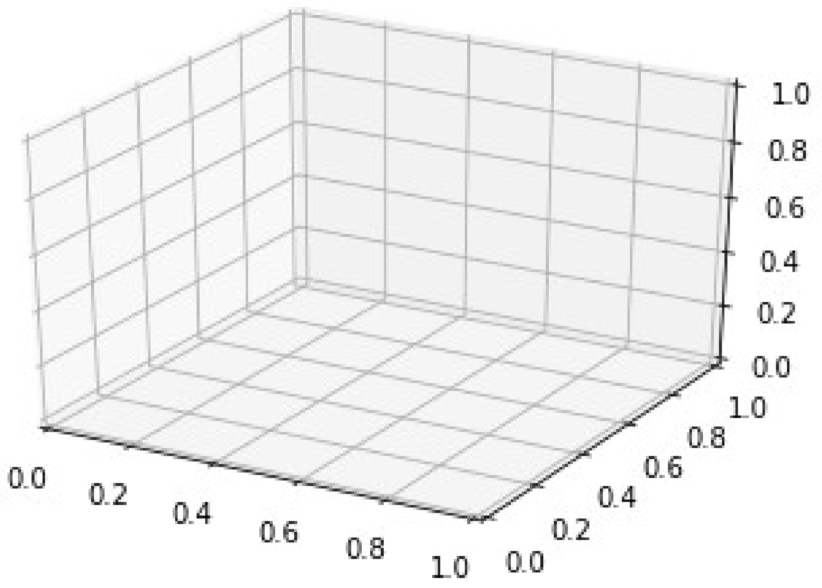
Matplotlib has basic level of support for three-dimensional plotting(3D plots). Adding one more dimension to 2D plots can help to visualize the data more effectively. To produce a 3D plot, first we have to create a 3D axes and then plot any of the 3D graphs.

3D plots in Matplotlib are enabled by importing the `mplot3d` toolkit, included with the main Matplotlib installation. It can be done as follows:-

```
from mpl_toolkits import mplot3d
```

Once the `mplot3d` submodule is imported, a 3D axes can be created by running the code below:-

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
```



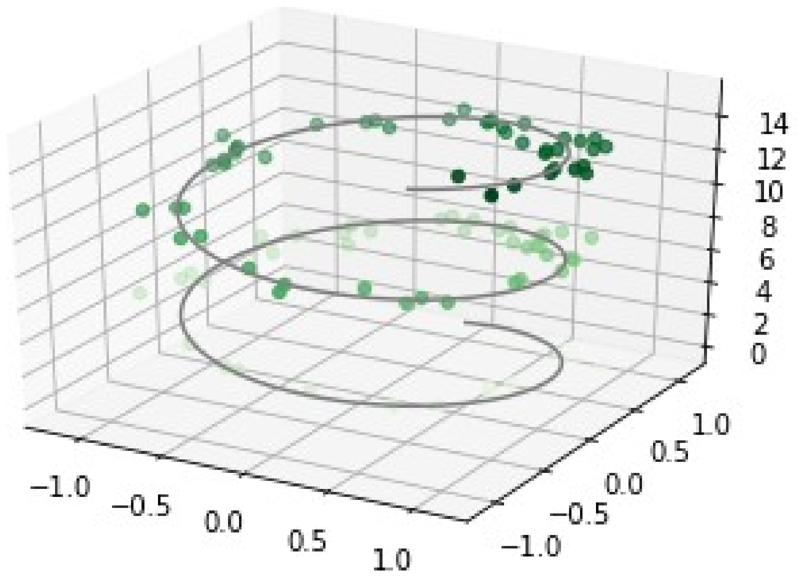
A 3D axes is drawn above. We can now plot variety of different plot types. If we want to view figures interactively rather than statistically, we can use `%matplotlib notebook` rather than `%matplotlib inline` when executing the above code.

3D Points and Lines

The most basic 3D Plot is a line or a collection of scatter plots. These plots can be created using the `ax.plot3D` and `ax.scatter3D` functions.

We can plot a 3D line or 3D scatter plot as follows:-

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

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

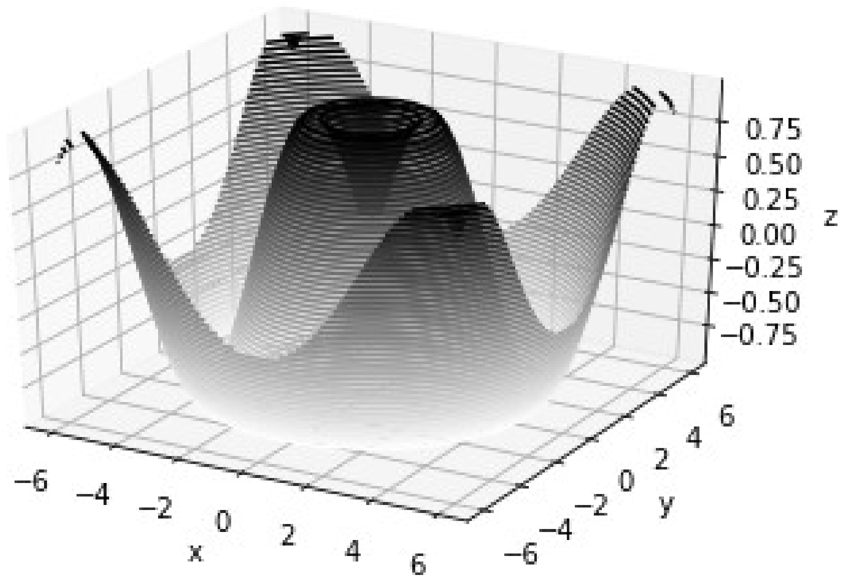


3D Contour Plots

3D Contour Plots are analogous to the 2D Contour Plots. `mplot3d` contains tools to create 3D plots usign the same inputs.

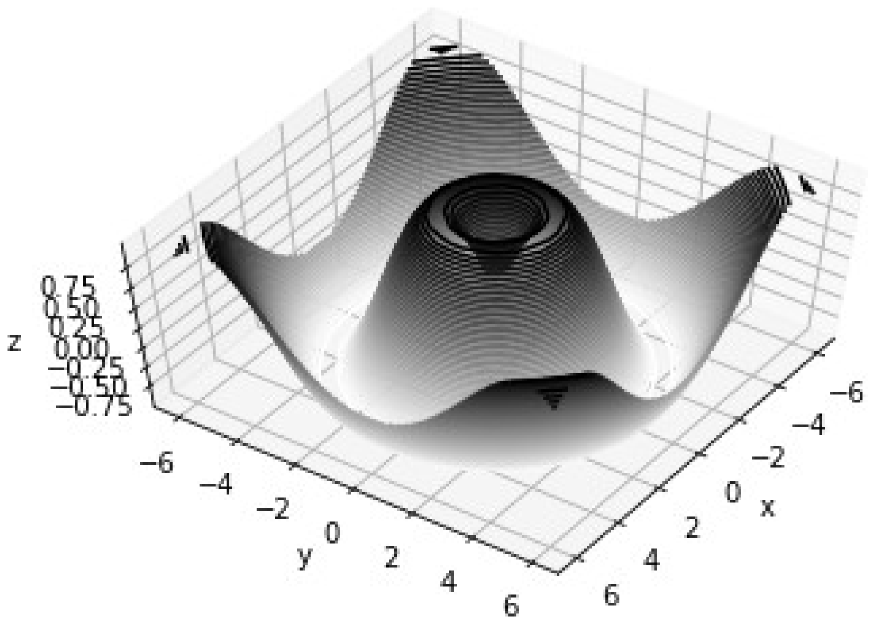
Like 2D `ax.contour` plots, `ax.contour3D` requires all the input data to be in the form of 2D regular grids, with the Z data evaluated at each point. I will show a 3D contour diagram of a 3D sinusoidal function.

```
x14 = np.linspace(-6, 6, 30)
y14 = np.linspace(-6, 6, 30)

X1,Y1 = np.meshgrid(x14, y14)

Z1 = np.sin(np.sqrt(X1 ** 2 + Y1 ** 2))

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X1, Y1, Z1, 50, cmap='binary')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



We can optimize the default view angle with `view_init` method. We can use this method to set the elevation and azimuthal angles as follows:-

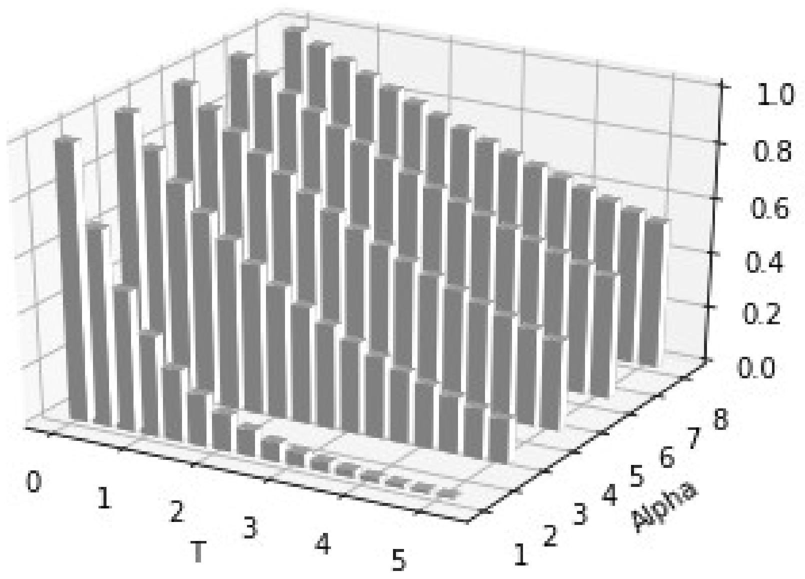
```
ax.view_init(60, 35)
fig
```



3D Bar Plot

Using several 2D layers in a 3D figure, we can plot multiple Bar Plots. However, we can also go full 3D and plot bar plots with actual 3D bars.

```
# Data generation
alpha = np.linspace(1, 8, 5)
t = np.linspace(0, 5, 16)
T, A = np.meshgrid(t, alpha)
data = np.exp(-T * (1. / A))


# Plotting
fig = plt.figure()
ax = fig.gca(projection = '3d')


Xi = T.flatten()
Yi = A.flatten()
Zi = np.zeros(data.size)


dx = .25 * np.ones(data.size)
dy = .25 * np.ones(data.size)
dz = data.flatten()


ax.set_xlabel('T')
ax.set_ylabel('Alpha')
ax.bar3d(Xi, Yi, Zi, dx, dy, dz, color = 'w')


plt.show()
```



31. Styles with Matplotlib Plots

We can view the list of all available styles by the following command.

`print(plt.style.availabe)`

```
# View list of all available styles

print(plt.style.available)
```

```
['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight',
'ggplot', 'grayscale', 'seaborn-bright', 'seaborn-colorblind',
'seaborn-dark-palette', 'seaborn-dark', 'seaborn-darkgrid', 'seaborn-
deep', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-
pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-
white', 'seaborn-whitegrid', 'seaborn', 'Solarize_Light2', 'tableau-
colorblind10', '_classic_test']
```

We can set the **Styles** for Matplotlib plots as follows:-

`plt.style.use('seaborn-bright')`

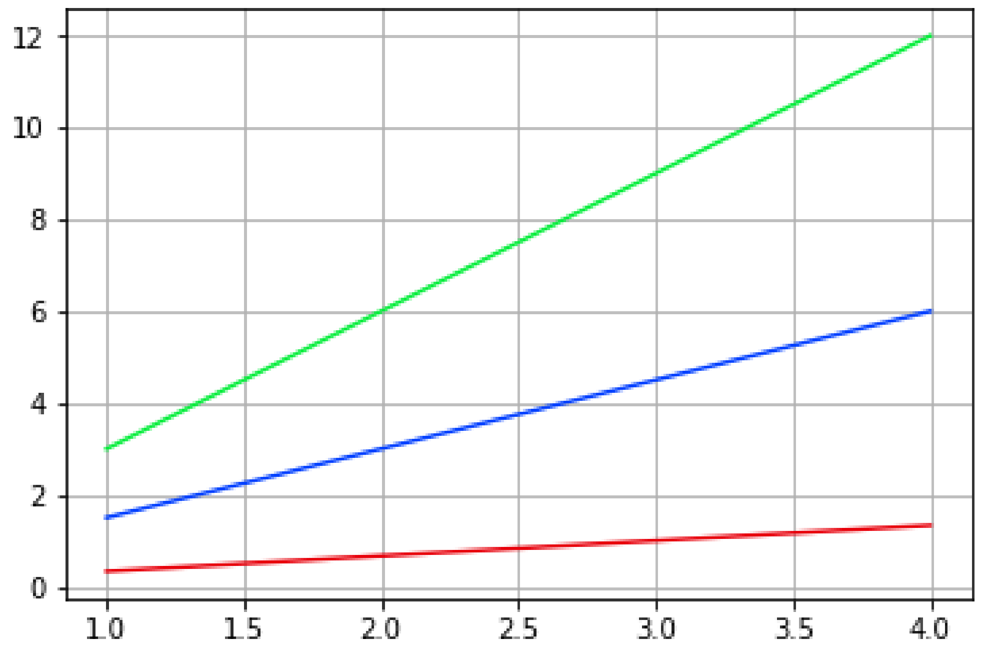
```
# Set styles for plots

plt.style.use('seaborn-bright')
```

I have set the **seaborn-bright** style for plots. So, the plot uses the **seaborn-bright** Matplotlib style for plots.

32. Adding a grid

In some cases, the background of a plot was completely blank. We can get more information, if there is a reference system in the plot. The reference system would improve the comprehension of the plot. An example of the reference system is adding a **grid**. We can add a grid to the plot by calling the **grid()** function. It takes one parameter, a Boolean value, to enable(if True) or disable(if False) the grid.

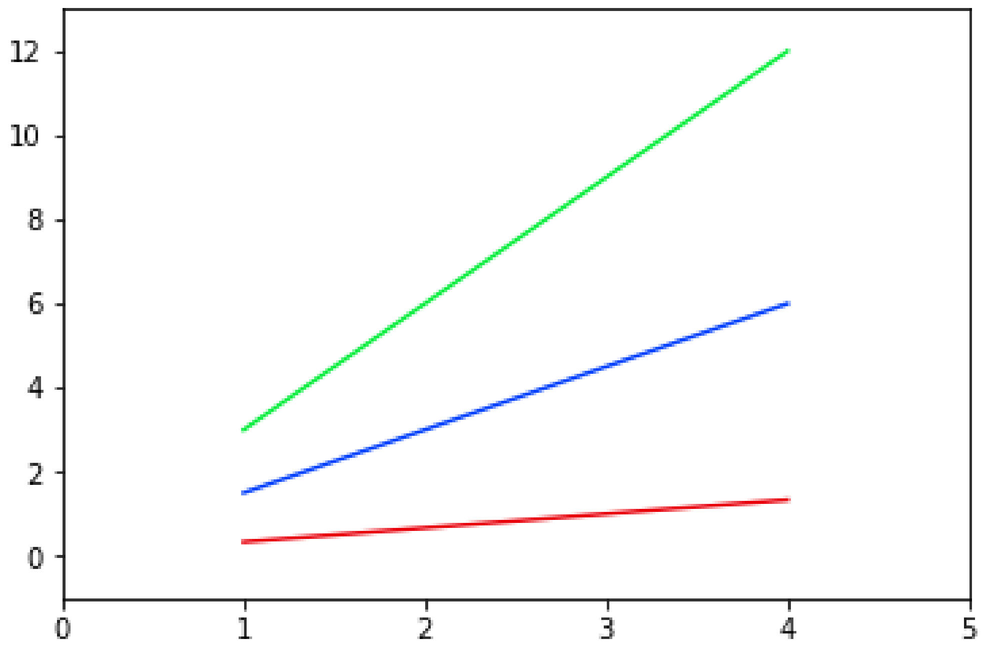
```
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)

plt.grid(True)

plt.show()
```



33. Handling axes

Matplotlib automatically sets the limits of the plot to precisely contain the plotted datasets. Sometimes, we want to set the axes limits ourself. We can set the axes limits with the **axis()** function as follows:-

```
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)

plt.axis()   # shows the current axis limits values

plt.axis([0, 5, -1, 13])

plt.show()
```



We can see that we now have more space around the lines.

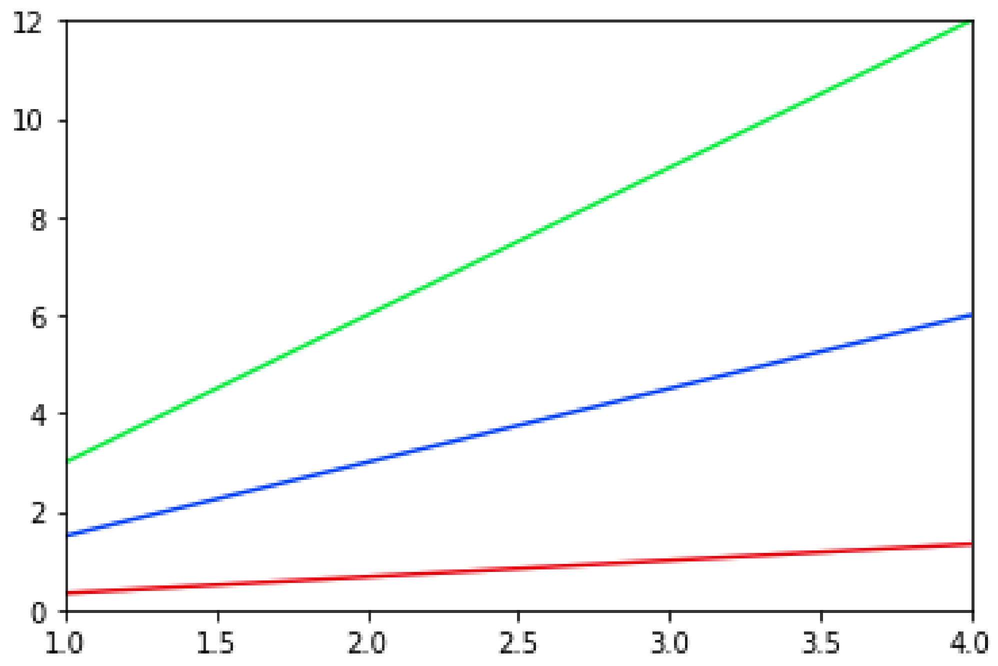
If we execute **axis()** without parameters, it returns the actual axis limits.

We can set parameters to **axis()** by a list of four values.

The list of four values are the keyword arguments [xmin, xmax, ymin, ymax] allows the minimum and maximum limits for X and Y axis respectively.

We can control the limits for each axis separately using the `xlim()` and `ylim()` functions. This can be done as follows:-

```
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)

plt.xlim([1.0, 4.0])

plt.ylim([0.0, 12.0])

(0.0, 12.0)
```



34. Handling X and Y ticks

Vertical and horizontal ticks are those little segments on the axes, coupled with axes labels, used to give a reference system on the graph.So, they form the origin and the grid lines.

Matplotlib provides two basic functions to manage them - **xticks()** and **yticks()**.

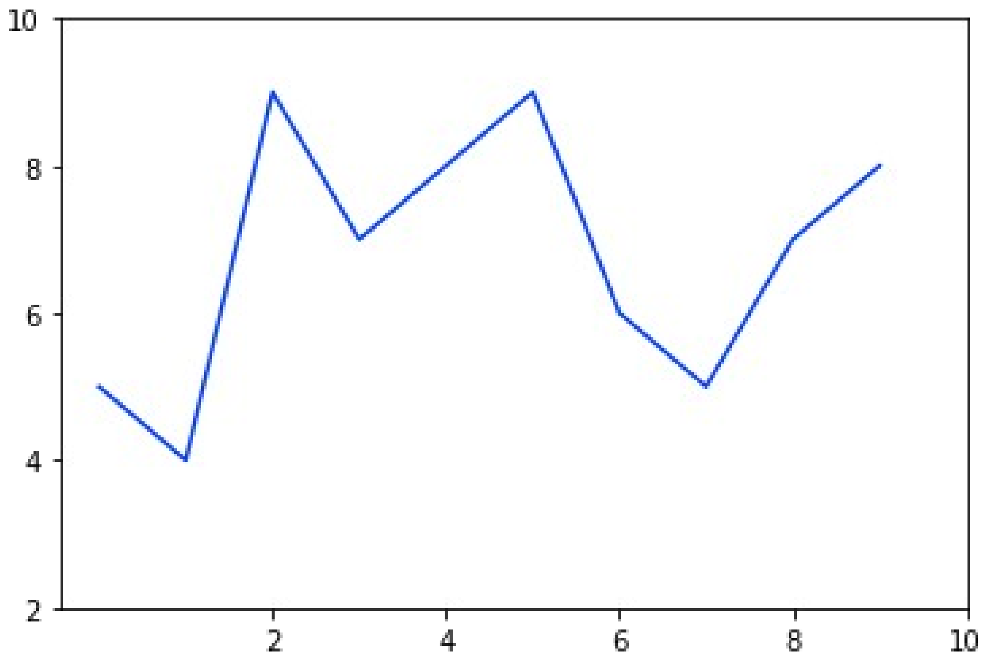
Executing with no arguments, the tick function returns the current ticks' locations and the labels corresponding to each of them.

We can pass arguments(in the form of lists) to the ticks functions. The arguments are:-

1. Locations of the ticks
2. Labels to draw at these locations.

We can demonstrate the usage of the ticks functions in the code snippet below:-

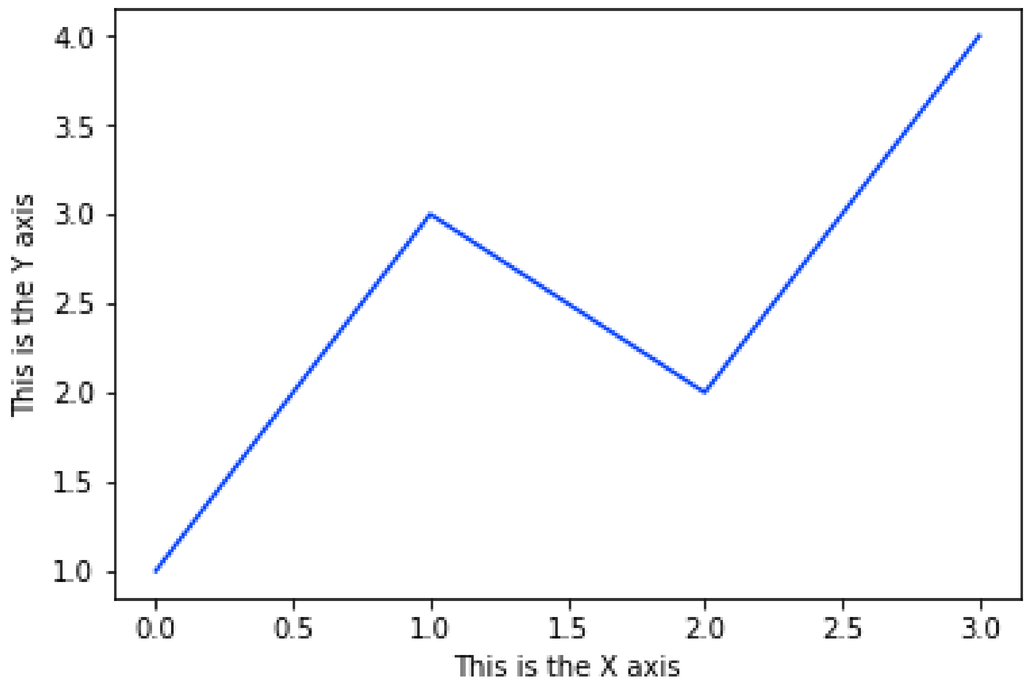
```
u = [5, 4, 9, 7, 8, 9, 6, 5, 7, 8]

plt.plot(u)

plt.xticks([2, 4, 6, 8, 10])
plt.yticks([2, 4, 6, 8, 10])

plt.show()
```



35. Adding labels

Another important piece of information to add to a plot is the axes labels, since they specify the type of data we are plotting.

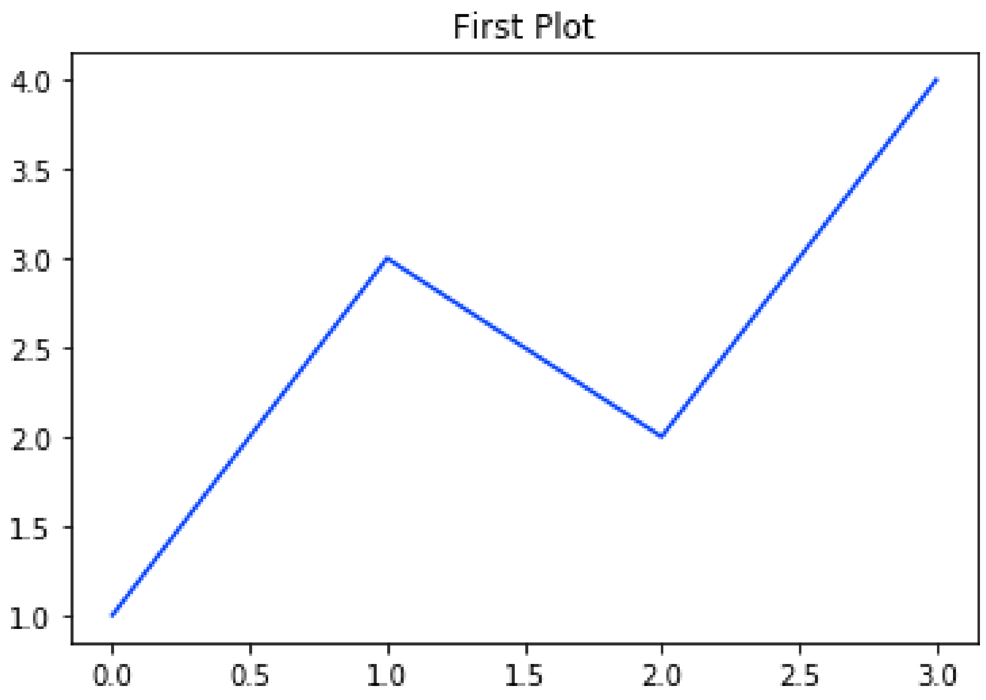
```
plt.plot([1, 3, 2, 4])

plt.xlabel('This is the X axis')

plt.ylabel('This is the Y axis')

plt.show()
```



36. Adding a title

The title of a plot describes about the plot. Matplotlib provides a simple function **title()** to add a title to an image.

```
plt.plot([1, 3, 2, 4])

plt.title('First Plot')

plt.show()
```



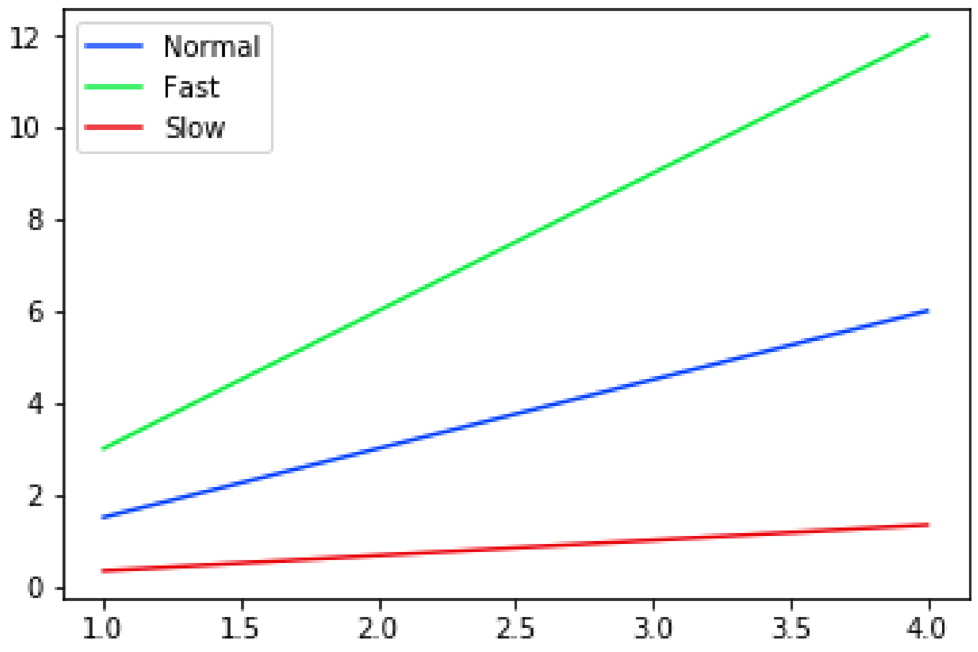
The above plot displays the output of the previous code. The title `First Plot` is displayed on top of the plot.

37. Adding a legend

Legends are used to describe what each line or curve means in the plot.

Legends for curves in a figure can be added in two ways. One method is to use the **legend** method of the axis object and pass a list/tuple of legend texts as follows:-

```
x15 = np.arange(1, 5)

fig, ax = plt.subplots()

ax.plot(x15, x15*1.5)
ax.plot(x15, x15*3.0)
ax.plot(x15, x15/3.0)

ax.legend(['Normal','Fast','Slow']);
```

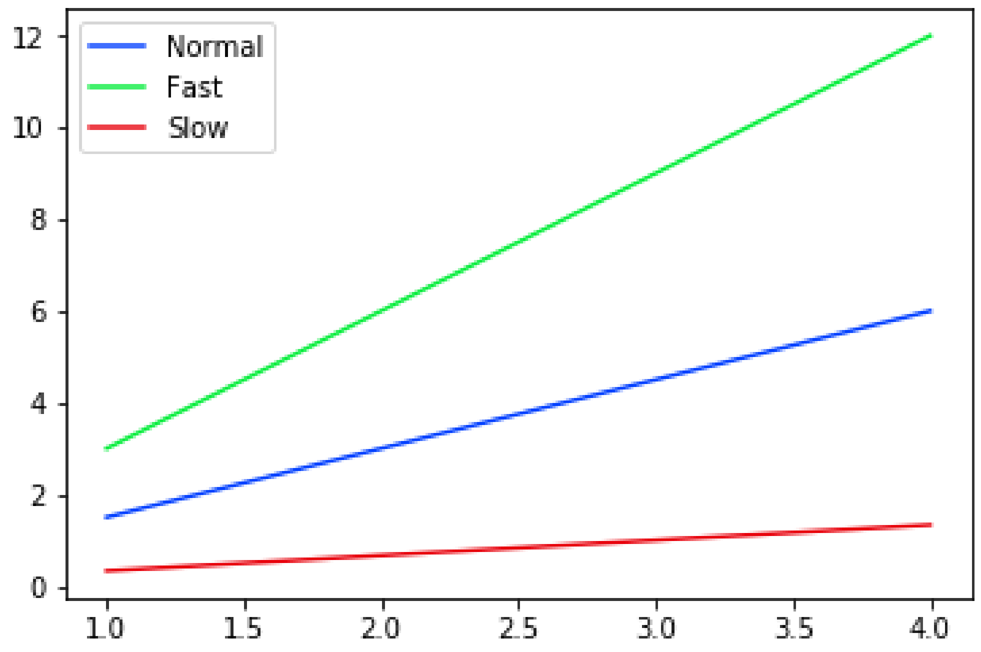


The above method follows the MATLAB API. It is prone to errors and unflexible if curves are added to or removed from the plot. It resulted in a wrongly labelled curve.

A better method is to use the **label** keyword argument when plots are added to the figure. Then we use the **legend** method without arguments to add the legend to the figure.

The advantage of this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly. It can be achieved by executing the code below:-

```
x15 = np.arange(1, 5)

fig, ax = plt.subplots()

ax.plot(x15, x15*1.5, label='Normal')
ax.plot(x15, x15*3.0, label='Fast')
ax.plot(x15, x15/3.0, label='Slow')

ax.legend();
```



The **legend** function takes an optional keyword argument **loc**. It specifies the location of the legend to be drawn. The **loc** takes numerical codes for the various places the legend can be drawn. The most common **loc** values are as follows:-

ax.legend(loc=0) # let Matplotlib decide the optimal location

ax.legend(loc=1) # upper right corner

ax.legend(loc=2) # upper left corner

ax.legend(loc=3) # lower left corner

ax.legend(loc=4) # lower right corner

ax.legend(loc=5) # right

ax.legend(loc=6) # center left

ax.legend(loc=7) # center right

ax.legend(loc=8) # lower center

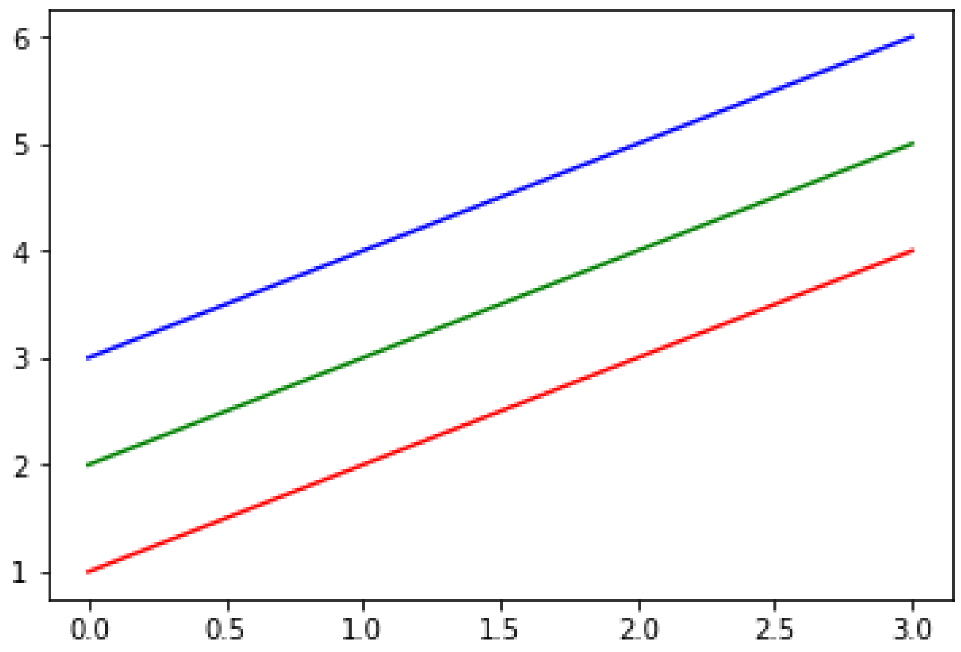
ax.legend(loc=9) # upper center

ax.legend(loc=10) # center

38. Control colours

We can draw different lines or curves in a plot with different colours. In the code below, we specify colour as the last argument to draw red, blue and green lines.

```
x16 = np.arange(1, 5)

plt.plot(x16, 'r')
plt.plot(x16+1, 'g')
plt.plot(x16+2, 'b')

plt.show()
```



The colour names and colour abbreviations is given in the following table:-

Colour abbreviation	**Colour name**
b	blue
c	cyan
g	green
k	black
m	magenta
r	red
w	white
y	yellow

There are several ways to specify colours, other than by colour abbreviations:

• The full colour name, such as yellow

• Hexadecimal string such as ##FF00FF

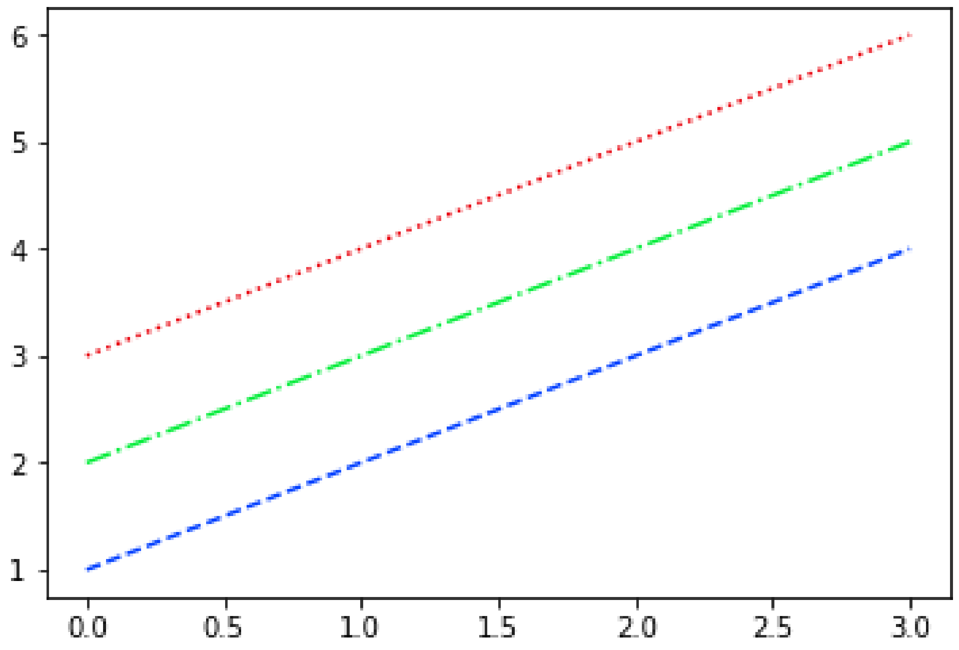
• RGB tuples, for example (1, 0, 1)

• Grayscale intensity, in string format such as ‘0.7’.

39. Control line styles

Matplotlib provides us different line style options to draw curves or plots. In the code below, I use different line styles to draw different plots.

```
x16 = np.arange(1, 5)

plt.plot(x16, '--', x16+1, '-.', x16+2, ':')

plt.show()
```



The above code snippet generates a blue dashed line, a green dash-dotted line and a red dotted line.

All the available line styles are available in the following table:

Style abbreviation Style

- `solid line`

-- dashed line

-. dash-dot line

: dotted line

Now, we can see the default format string for a single line plot is 'b-'.

40. Summary