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:-

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 - R PROGRAMING PACKAGE

• ggplot - R PROGRAMING PACKAGE

• pygal - R programing pacakge

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

In [1]:
```
#Import Dependencies
import numpy as np
import pandas as pd
```

In [2]:
```
# 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.

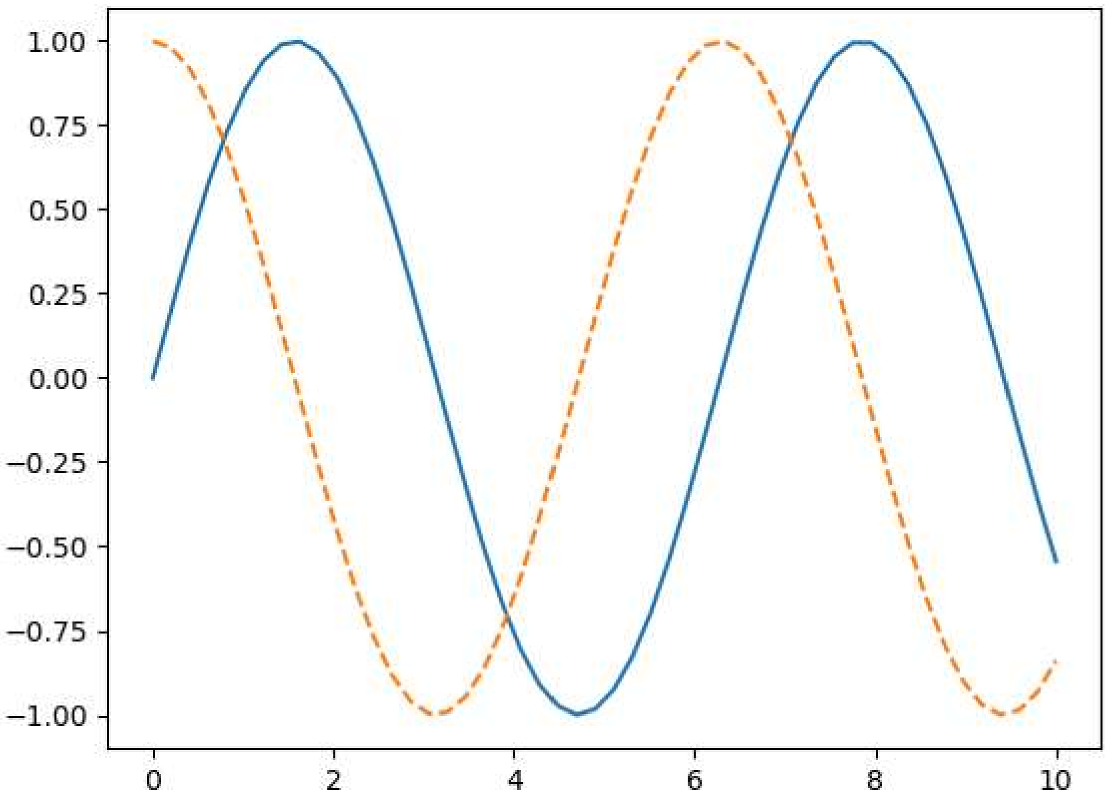
Interactive plotting within a Jupyter Notebook can be done with the **%matplotlib** command. There are two possible options to work with graphics in Jupyter Notebook. These are as follows:-

• **%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.

In [3]:
```
%matplotlib inline
x1 = np.linspace(0,10,50)

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

plt.plot(x1,np.sin(x1),'-')#plots a sine curve using a solid line ('-').which is de
plt.plot(x1,np.cos(x1), '--')
#plt.plot(x1,np.tan(x1), '-')

plt.show()
```



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.

In [4]:
```
#get current figure
plt.gcf()
```

Out[4]: <Figure size 640x480 with 0 Axes>

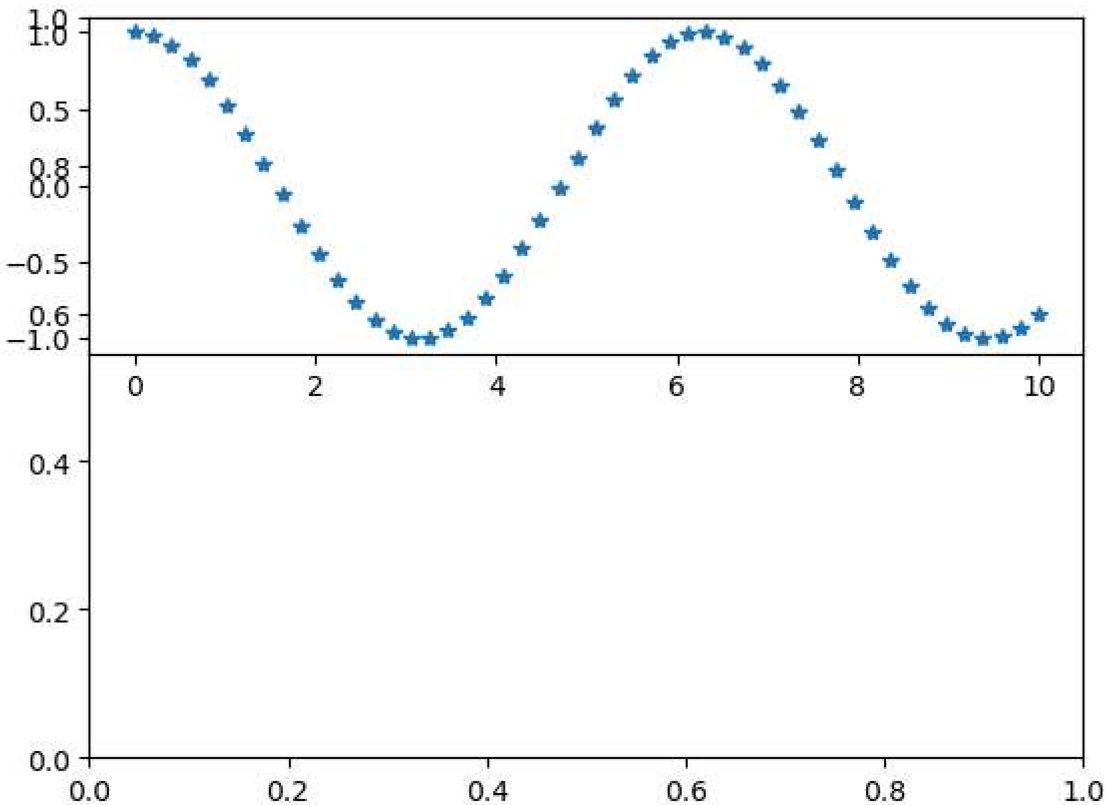
In [5]:
```
#get current axes
plt.gca()
```

Out[5]: <Axes: >

In [6]:
```
#create the first of two panels and set current axis

plt.subplot(2,1,1)# (rows, columns, panel number)
plt.plot(x1,np.cos(x1), '*')
```

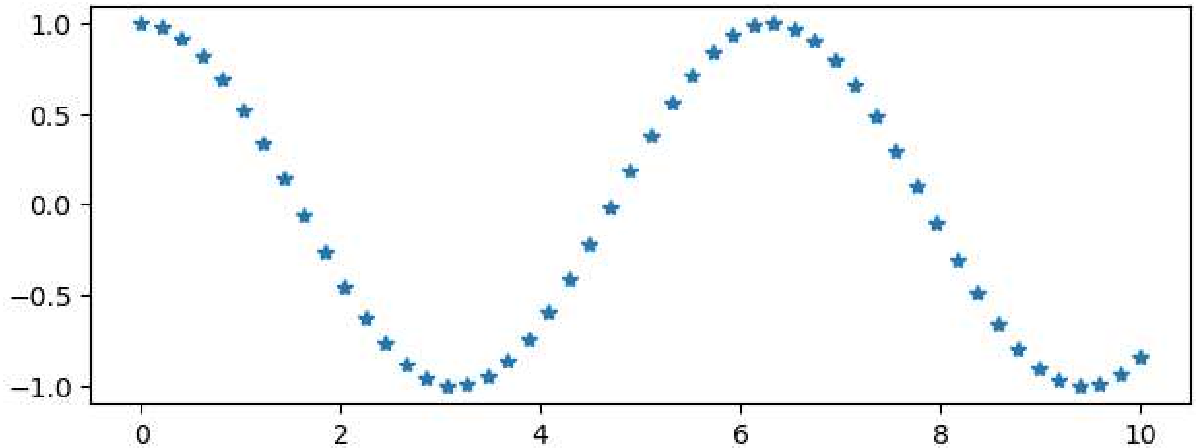
Out[6]: [<matplotlib.lines.Line2D at 0x1cffc49bd10>]

In [7]:
```
plt.show()
```



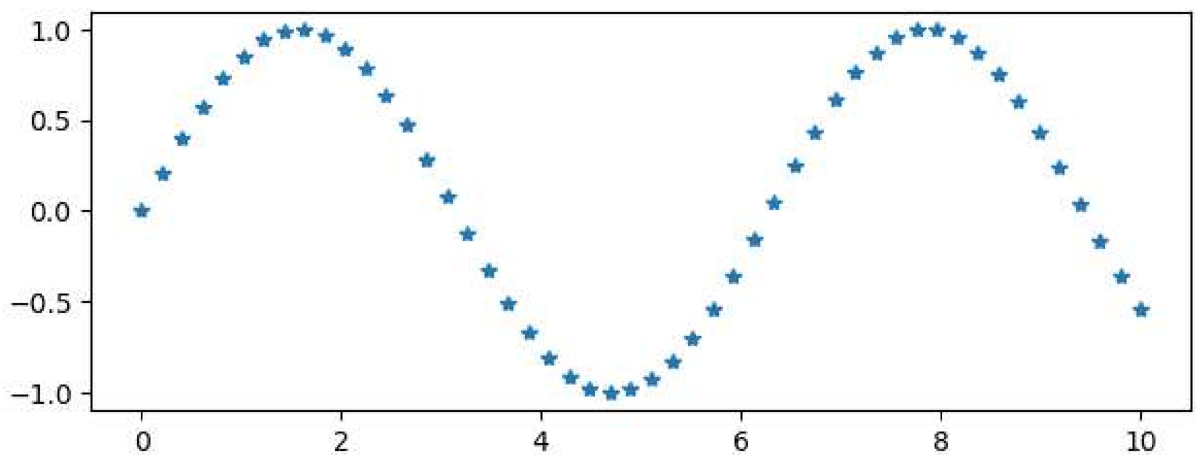
In [8]:
```
#Use plt.tight_layout() at the end to avoid overlapping of titles and labels.

plt.subplot(2,1,1)# (rows, columns, panel number)
plt.plot(x1,np.cos(x1), '*')
plt.tight_layout()
```

In [9]:
```
plt.show()
```



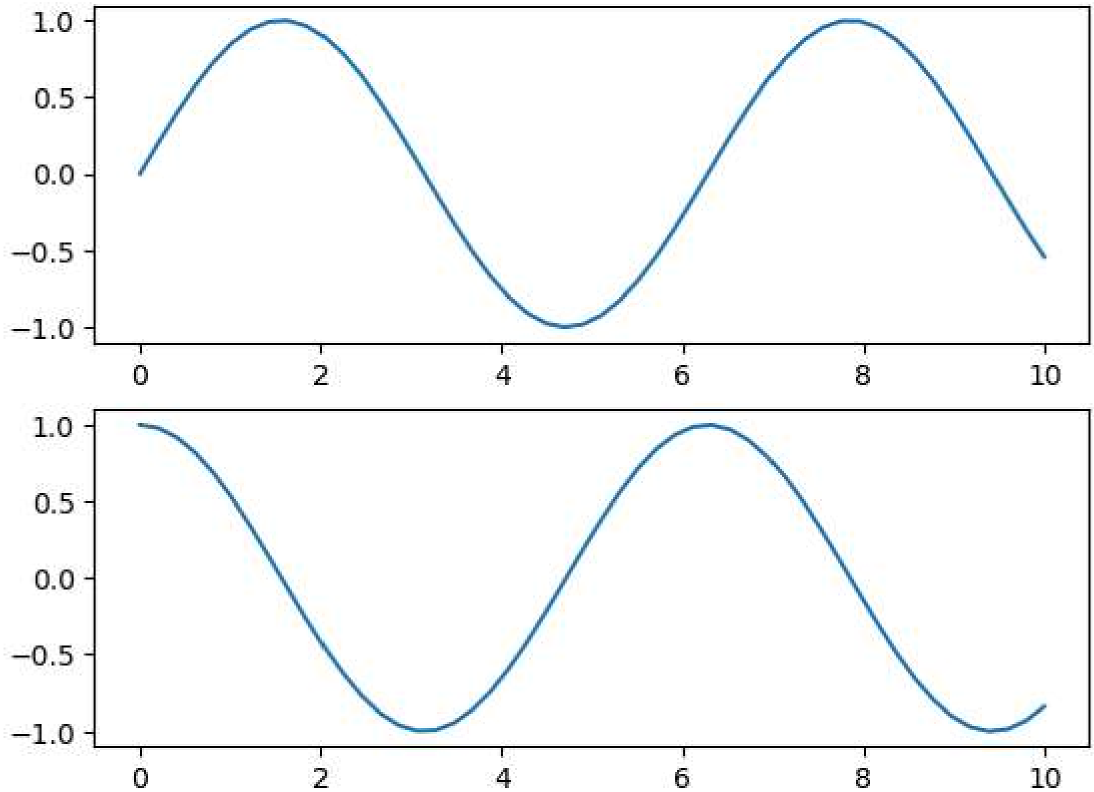
In [10]:
```
#Use plt.tight_layout() at the end to avoid overlapping of titles and labels.

plt.subplot(2,1,2)# (rows, columns, panel number)
plt.plot(x1,np.sin(x1), '*')
plt.tight_layout()
```

In [11]:
```
plt.show()
```



In [12]:
```
#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))

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

In [13]:
```
plt.show()
```



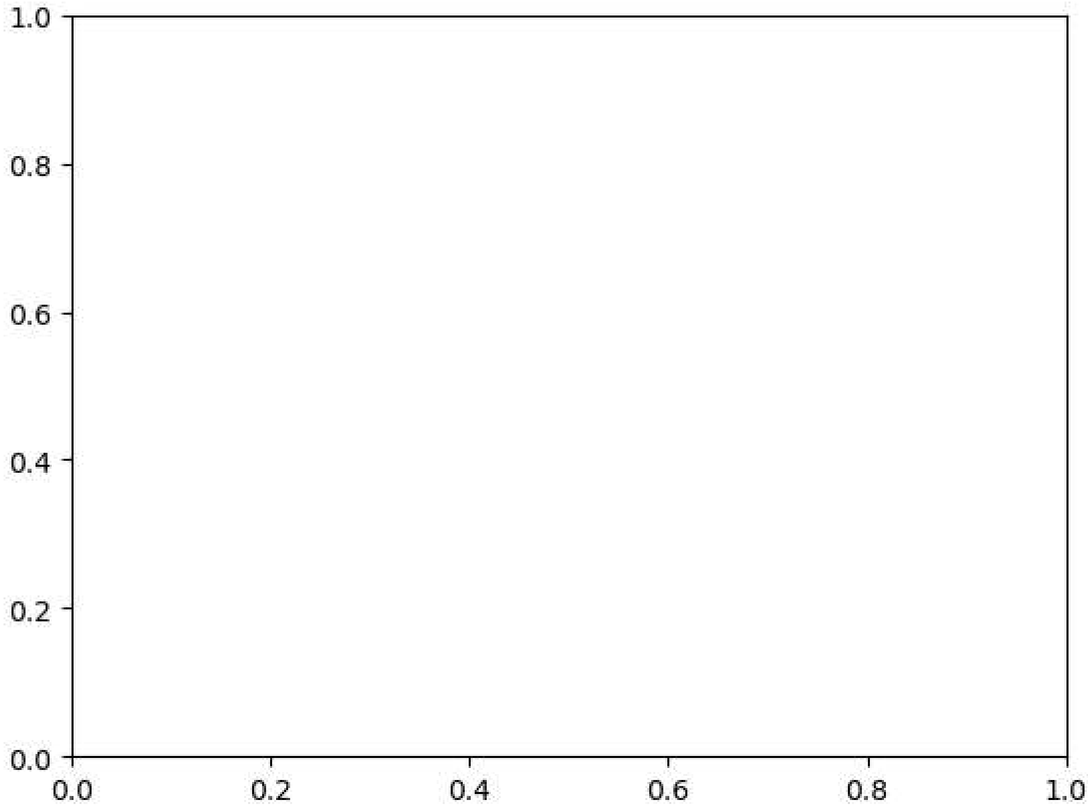
In [14]:
```
#get current figure information

print(plt.gcf())
```

```
Figure(640x480)
```

In [15]:
```
#get current axis information
print(plt.gca())
```

```
Axes(0.125,0.11;0.775x0.77)
```

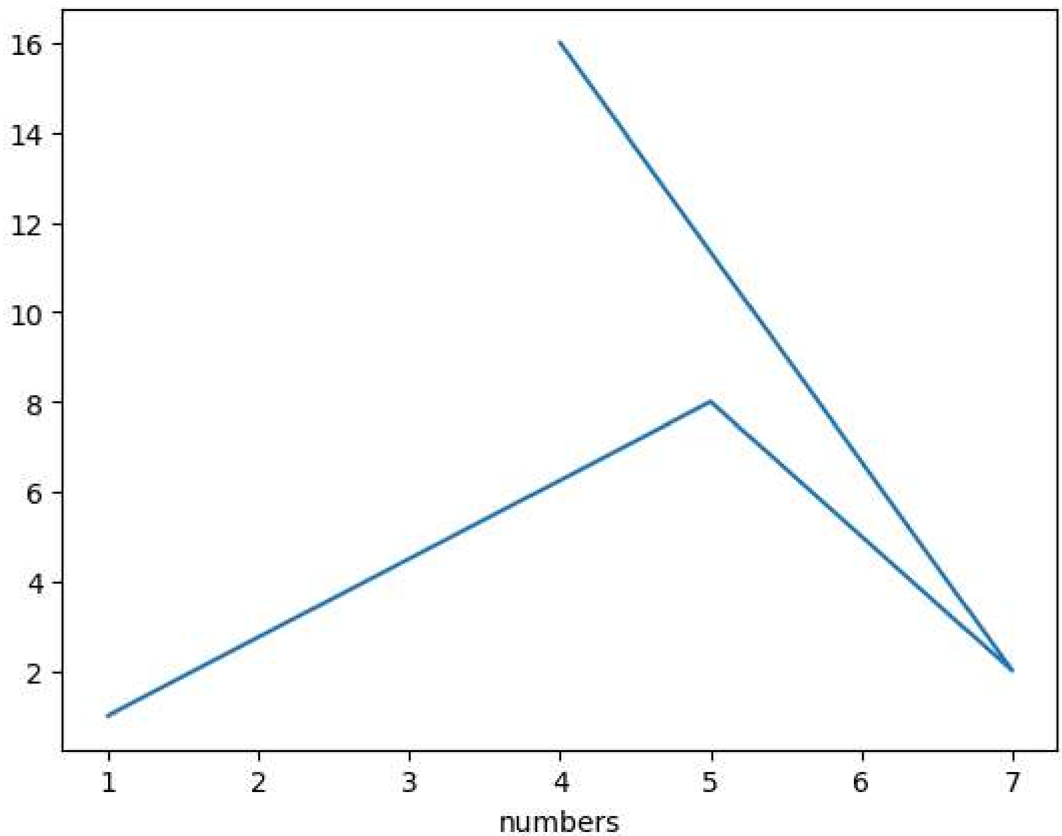
In [16]:
```
plt.show()
```



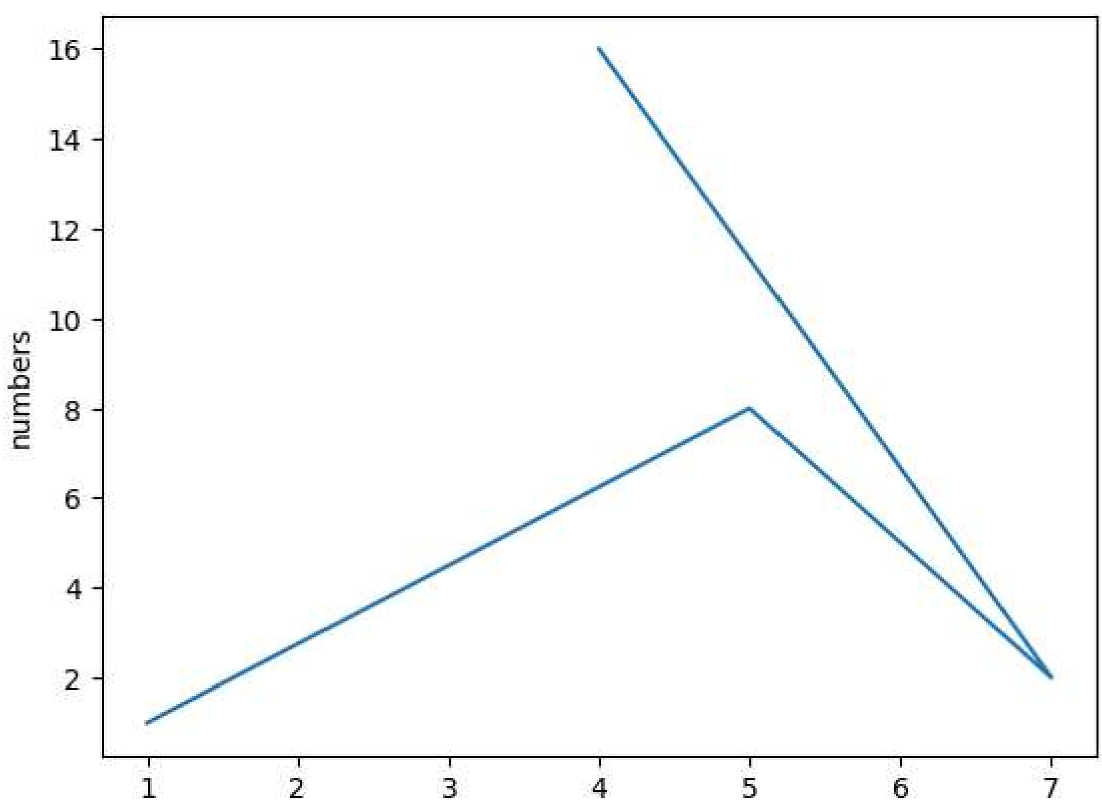
Visualization with Pyplot

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].

In [17]:
```
plt.plot([1,5,7,4],[1,8,2,16])
plt.xlabel('numbers')
plt.show()
```



In [18]:
```
plt.plot([1,5,7,4],[1,8,2,16])
plt.ylabel('numbers')
plt.show()
```



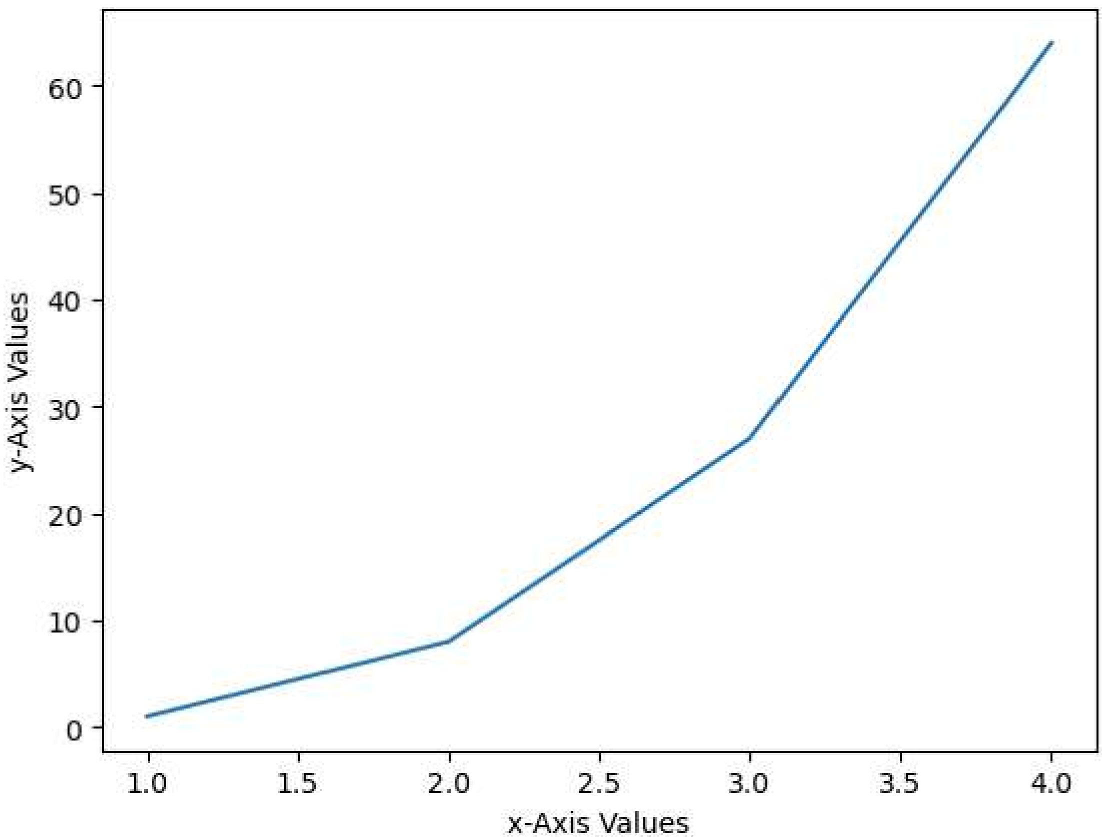
In [19]:
```
#how to see the difference between x & y axis values
print(plt.xlabel("x-Axis Values"))
print(plt.ylabel("y-Axis Values"))
```

```
Text(0.5, 0, 'x-Axis Values')
Text(0, 0.5, 'y-Axis Values')
```

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:-

In [20]:
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4],[1,8,27,64])
plt.show()
```

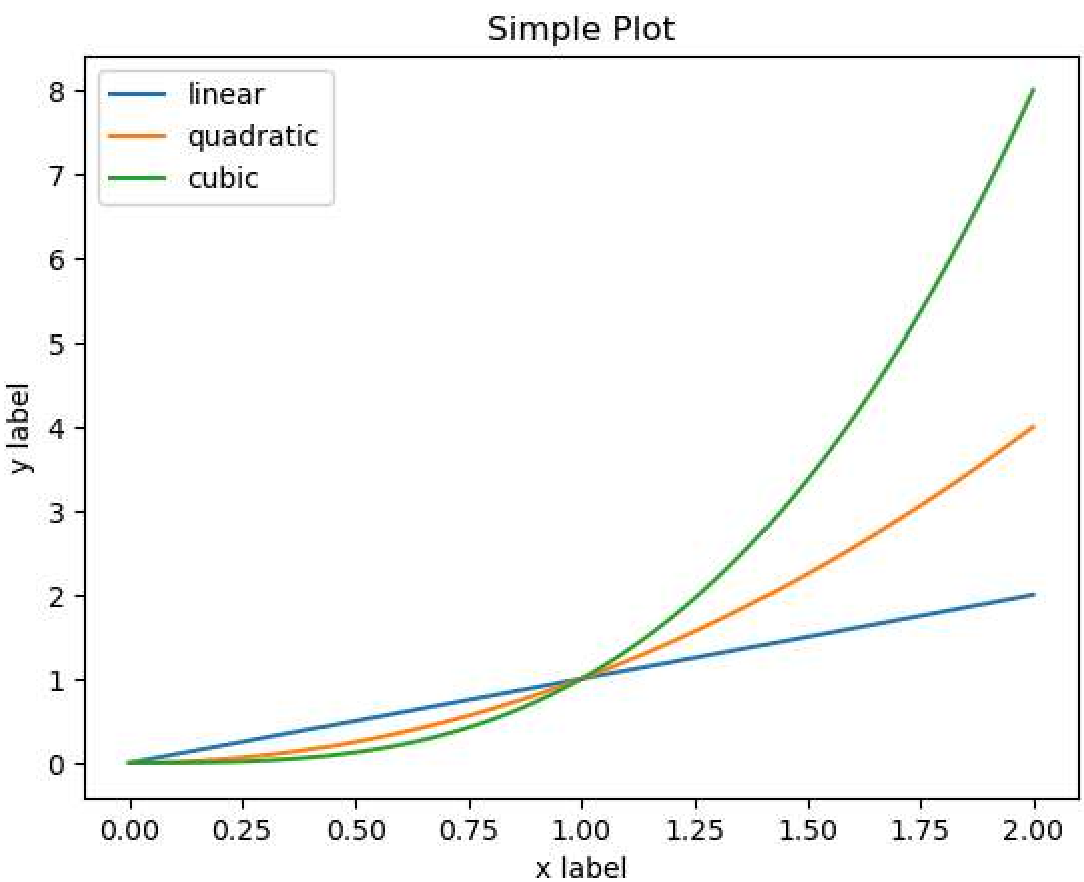


State-machine interface

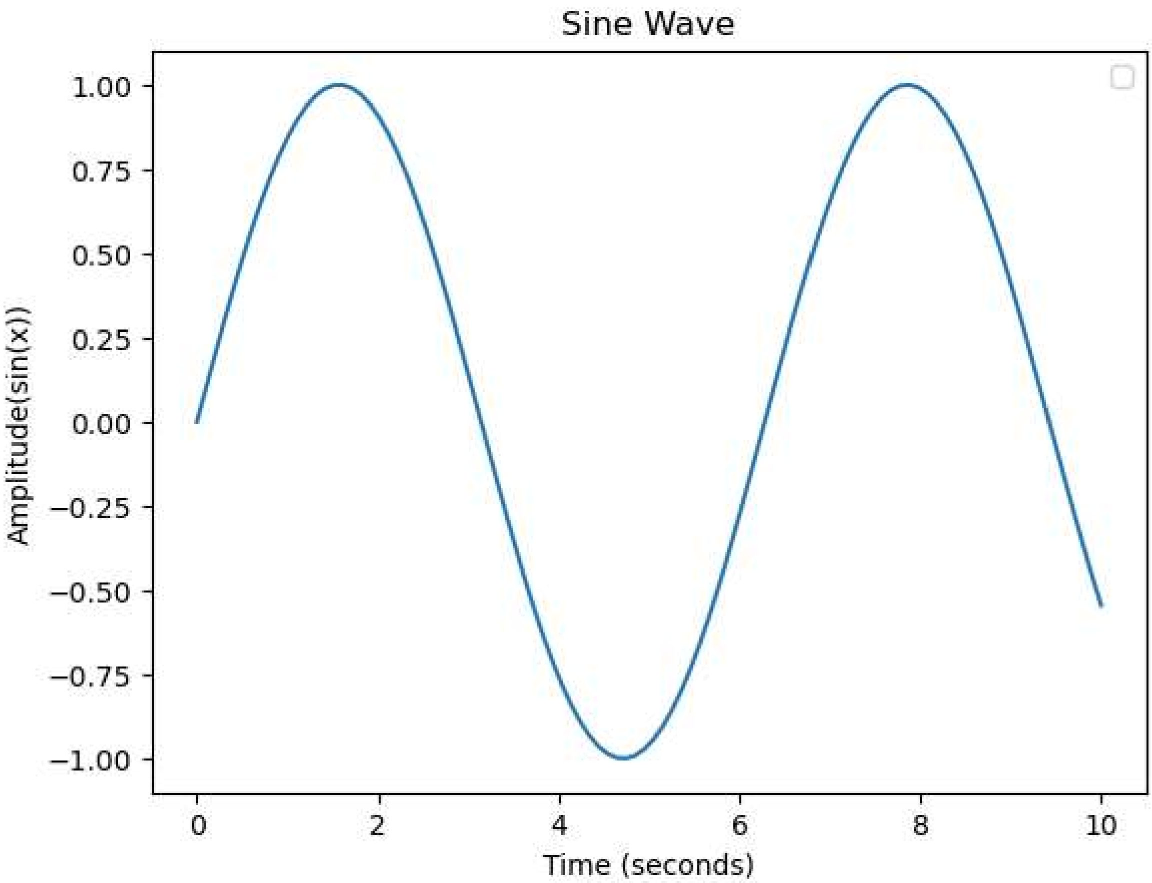
Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot. For example:

In [22]:
```
x=np.linspace(0,2,100)

# Plotting multiple functions
plt.plot(x,x,label='linear')# y = x → straight line
```

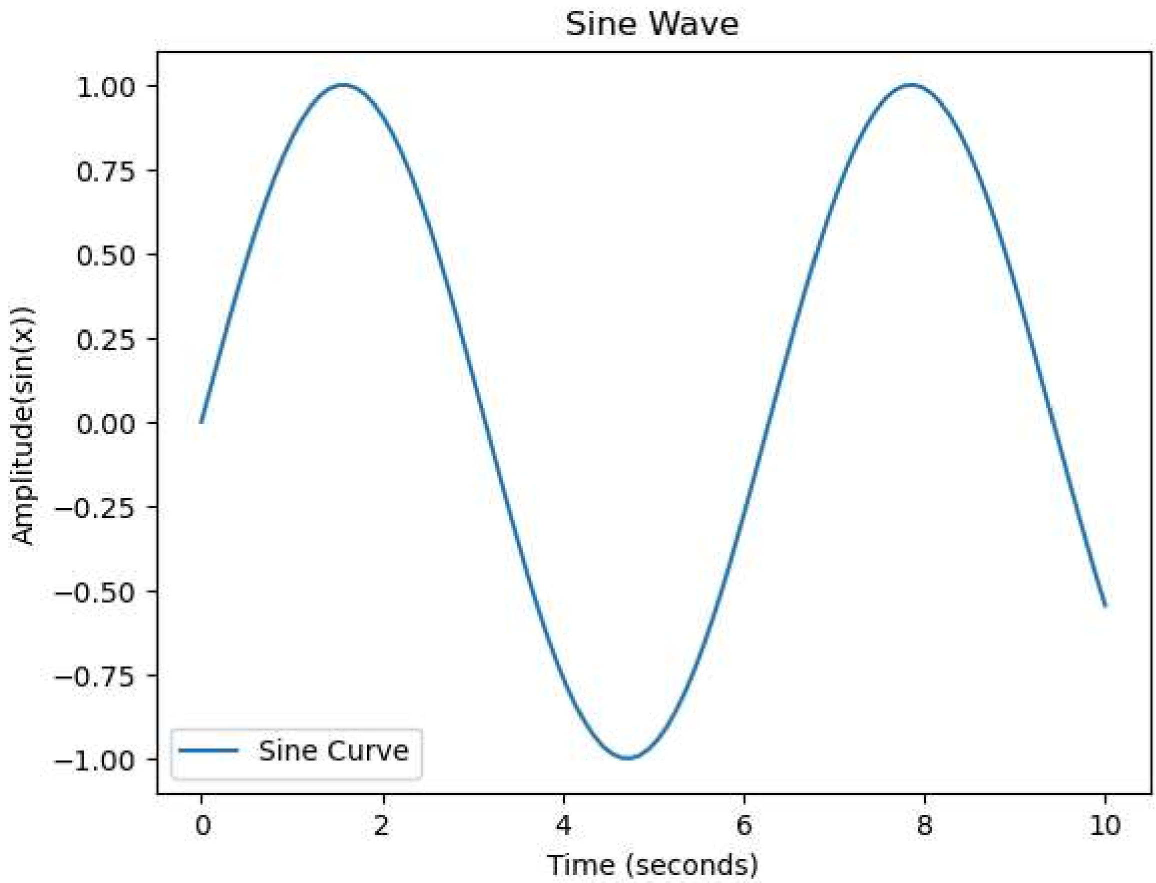
```
plt.plot(x,x**2,label='quadratic')# y = x² → parabolic curve
plt.plot(x,x**3,label='cubic')# y = x³ → steep curve

# Labeling the axes(They are used to describe what the X and Y axes represent,
#making your plot more readable and meaningful to viewers.)
plt.xlabel('x label')
plt.ylabel('y label')

# Title of the plot
plt.title("Simple Plot")

# Add a legend (to show which line is which)
plt.legend()

plt.show()
```



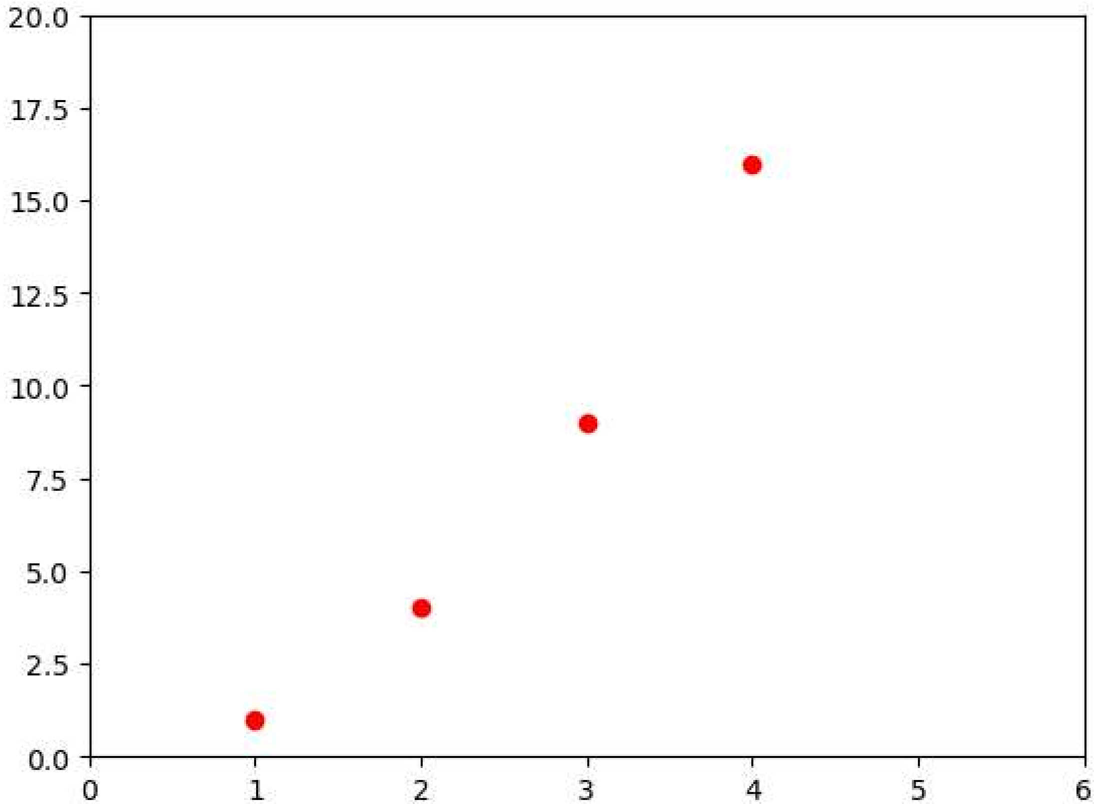
In [23]:
```
import warnings
warnings.filterwarnings('ignore')

x= np.linspace(0,10,100)
y=np.sin(x)

plt.plot(x,y)
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude(sin(x))")
plt.title("Sine Wave")
plt.legend()#not worked cause label didn't assign
plt.show()
```



In [24]:
```
x= np.linspace(0,10,100)
y=np.sin(x)

plt.plot(x,y,label='Sine Curve')#label assigned
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude(sin(x))")
plt.title("Sine Wave")
plt.legend()
plt.show()
```

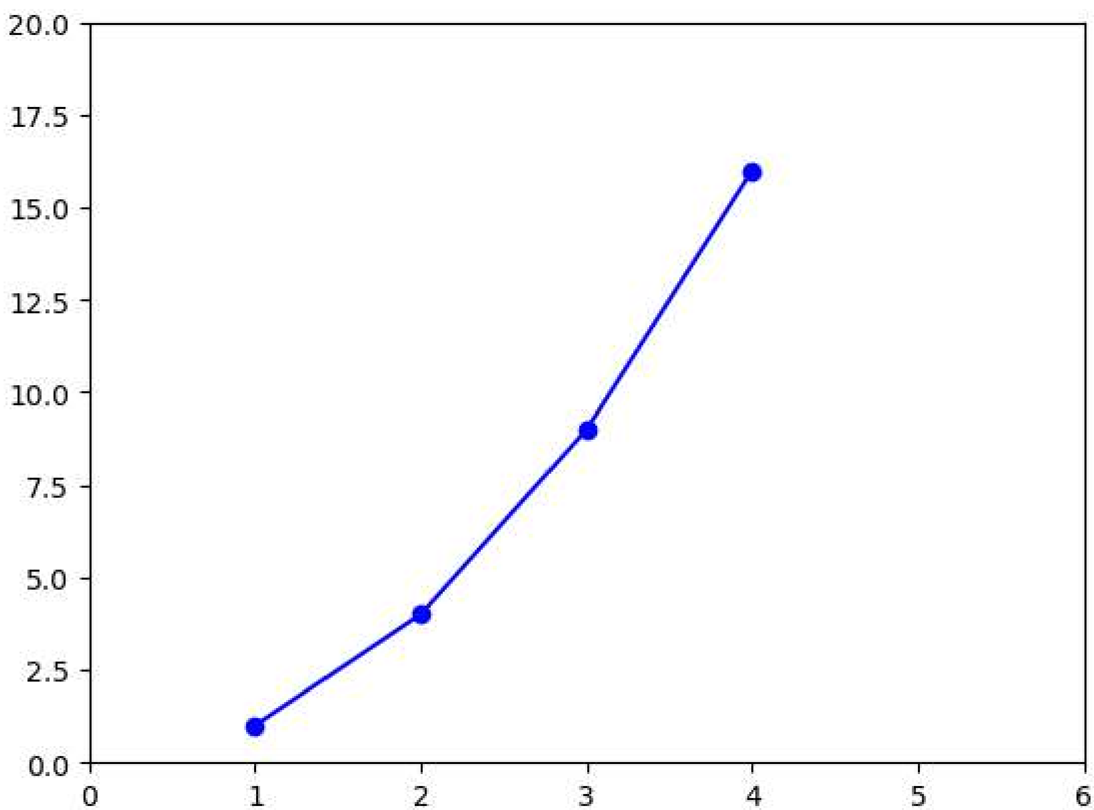


In [25]:
```
plt.plot([1,2,3,4],[1,4,9,16],'or')#o is Circle marks & r is red color
plt.axis([0,6,0,20])
plt.show()
```



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

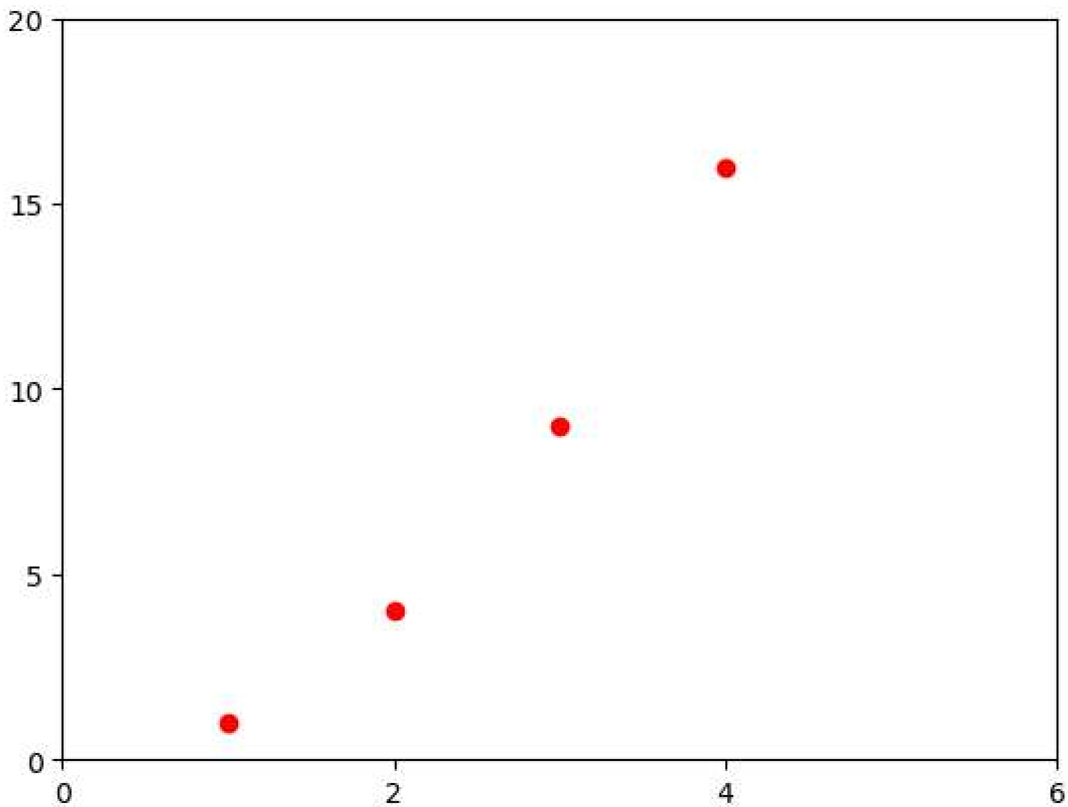
In [26]:
```
plt.plot([1,2,3,4],[1,4,9,16],'b-o')#o is Circle marks & b is blue color
plt.axis([0,6,0,20])
plt.show()
```



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:-

In [27]:
```
plt.plot([1,2,3,4],[1,4,9,16],'or')
plt.axis([0,6,0,20])
plt.yticks([0,5,10,15,20])#set custom y axis ticks
plt.xticks([0,2,4,6]) #optional: set custom x-axis ticks
plt.show()
```



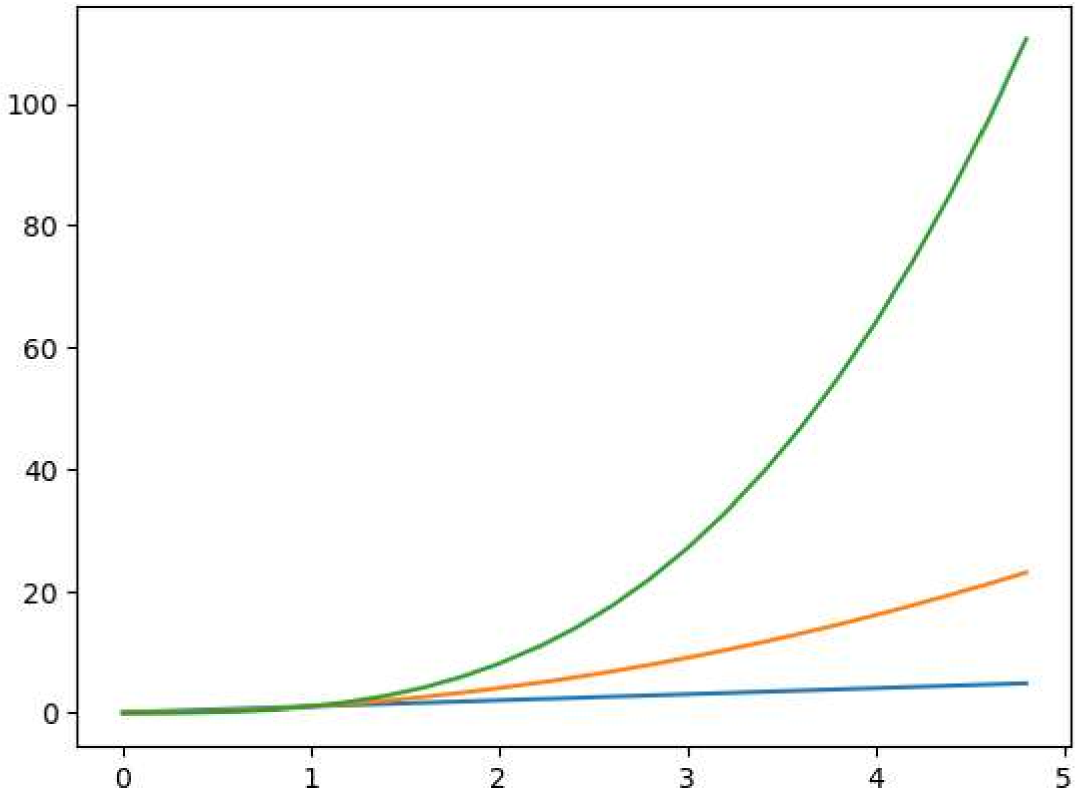
The **axis()** command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

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.

In [29]:
```
#evenly sampled time at 200ms intervals
```

```
t = np.arange(0.,5.,0.2)#start,step,stop

plt.plot(t,t,t,t**2,t,t**3,)
plt.show()
```



OBJECTT_ ORIENTED API

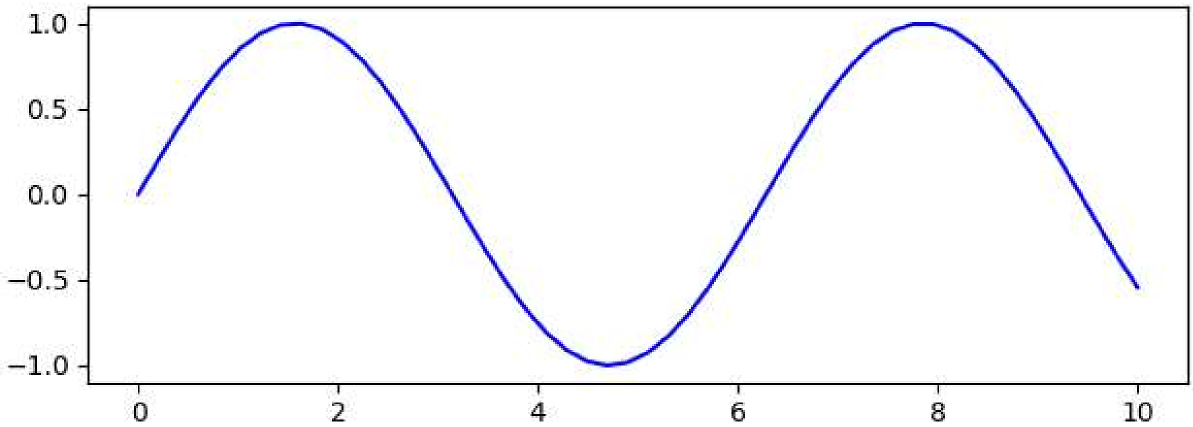
The Object-Oriented API is available for more complex plotting situations. It allows us to exercise more control over the figure. In Pyplot API, we depend on some notion of an "active" figure or axes. But, in the Object-Oriented API the plotting functions are methods of explicit Figure and Axes objects.

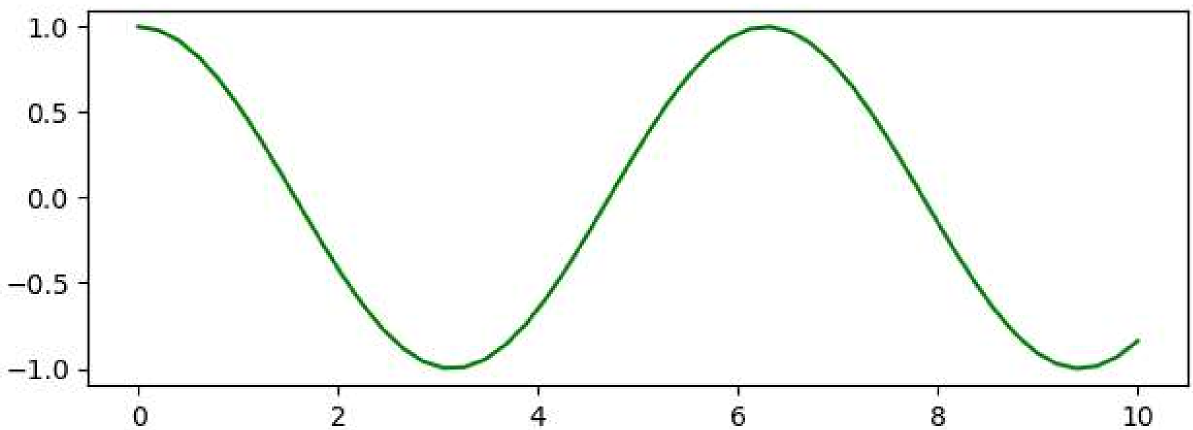
Figure is the top level container for all the plot elements. We can think of the Figure object as a box-like container containing one or more Axes.

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

In [30]:
```
# first create a grid of plots (2 rows, 1 column by default)
#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), 'g-');

plt.tight_layout()#add spacing
```

In [31]:
```
plt.show()
```





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:-

In [32]:
```
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',label = 'line x2')

plt.legend()


axes.set_xlabel('x2')
axes.set_ylabel('y2')
```

```
axes.set_title('title');

#plt.tight_layout()
```

In [33]:
```
plt.show()
```

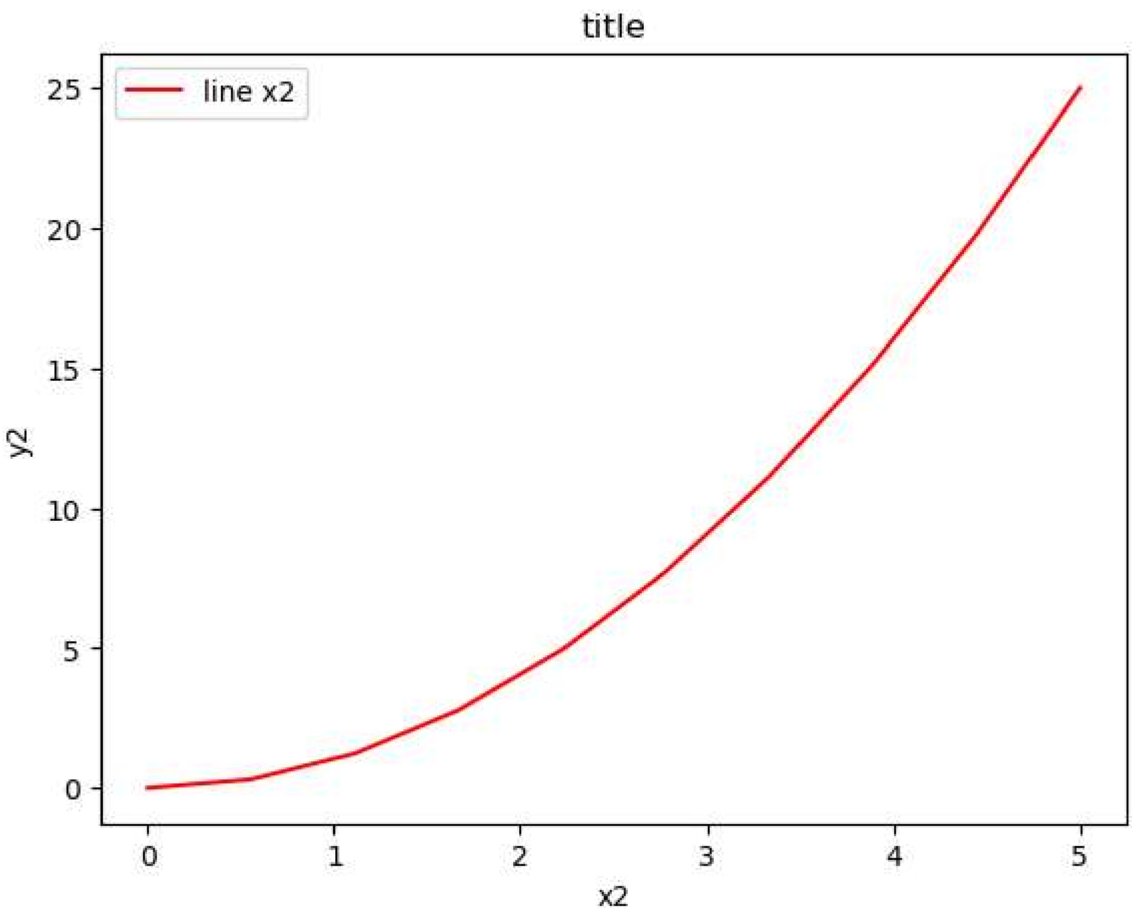


Figure and Axes I start by creating a figure and an axes. A figure and axes can be created as follows:

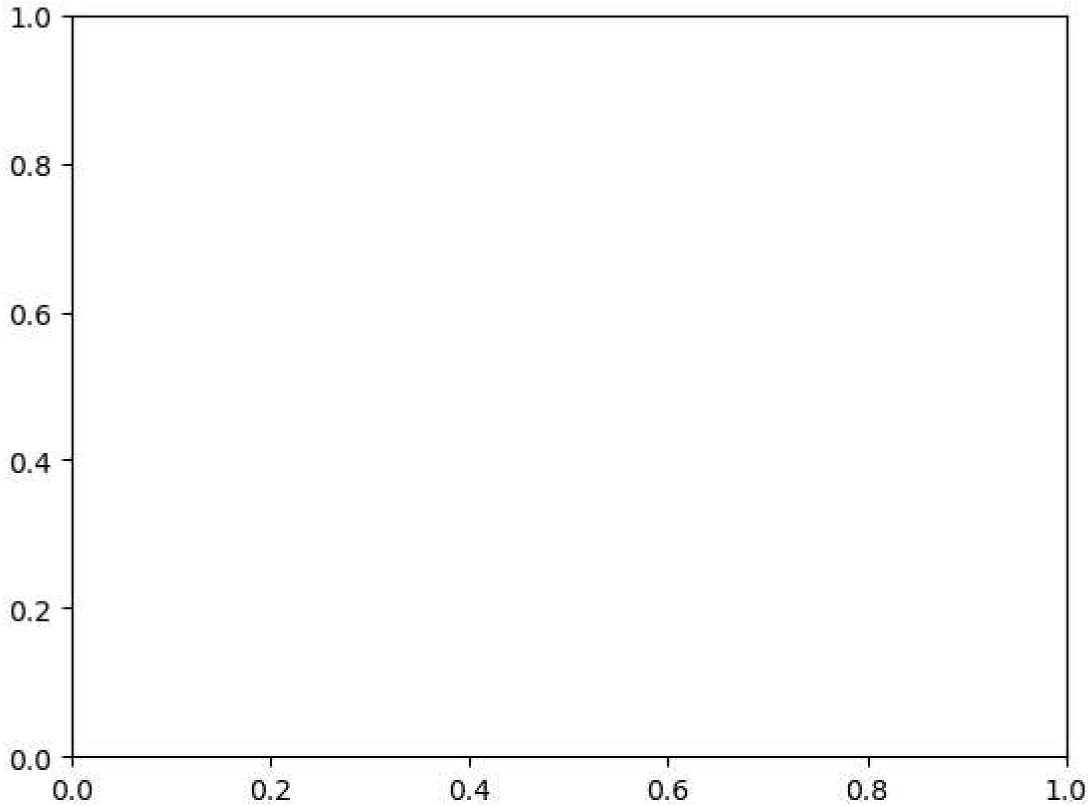
fig = plt.figure()

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.

In [34]:
```
fig = plt.figure()
ax = plt.axes()
```

In [35]:
```
plt.show()
```



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)

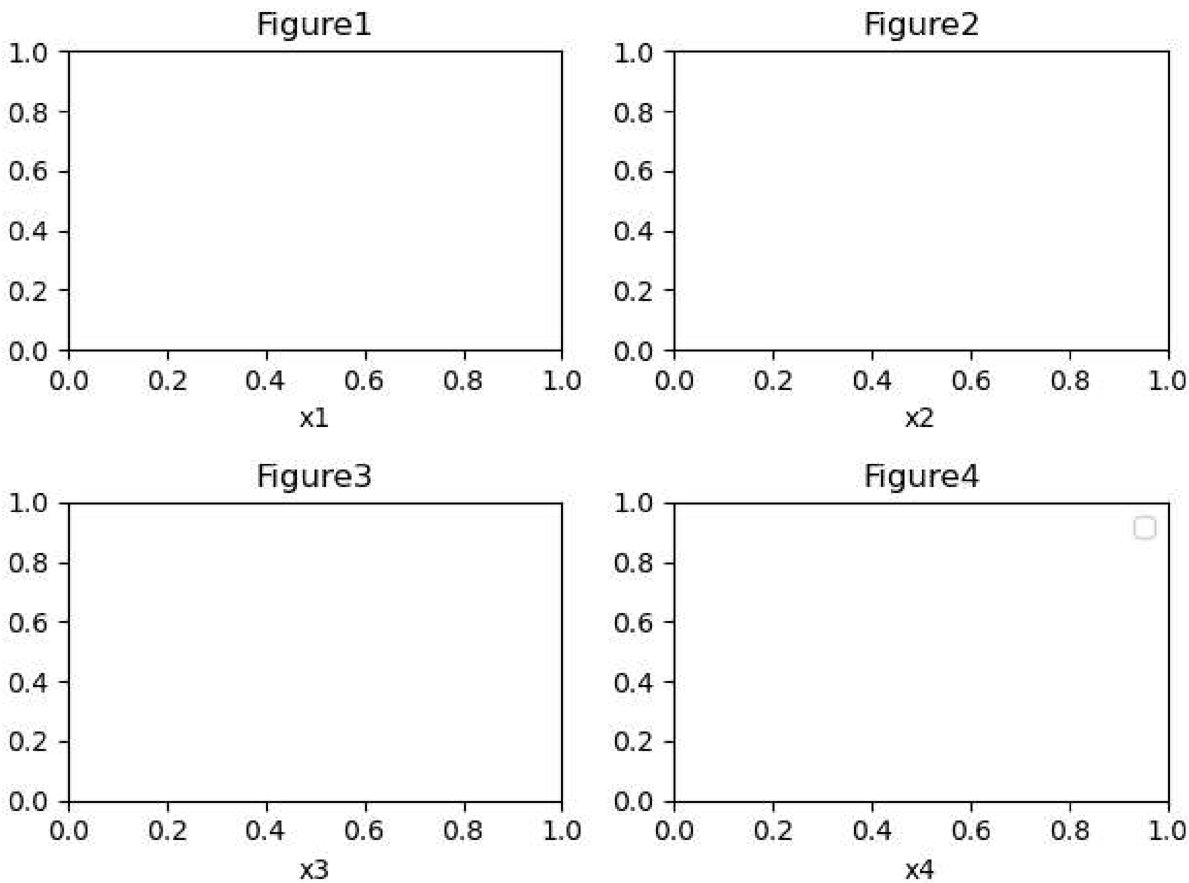
ax3 = fig.add_subplot(2, 2, 3)

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

In [37]:
```
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)

ax1.set_xlabel('x1')
ax2.set_xlabel('x2')
ax3.set_xlabel('x3')
ax4.set_xlabel('x4')

ax1.set_title('Figure1')
ax2.set_title('Figure2')
ax3.set_title('Figure3')
ax4.set_title('Figure4')

plt.legend()#it will not work cause plot line is not associate with a label.

plt.tight_layout()
```

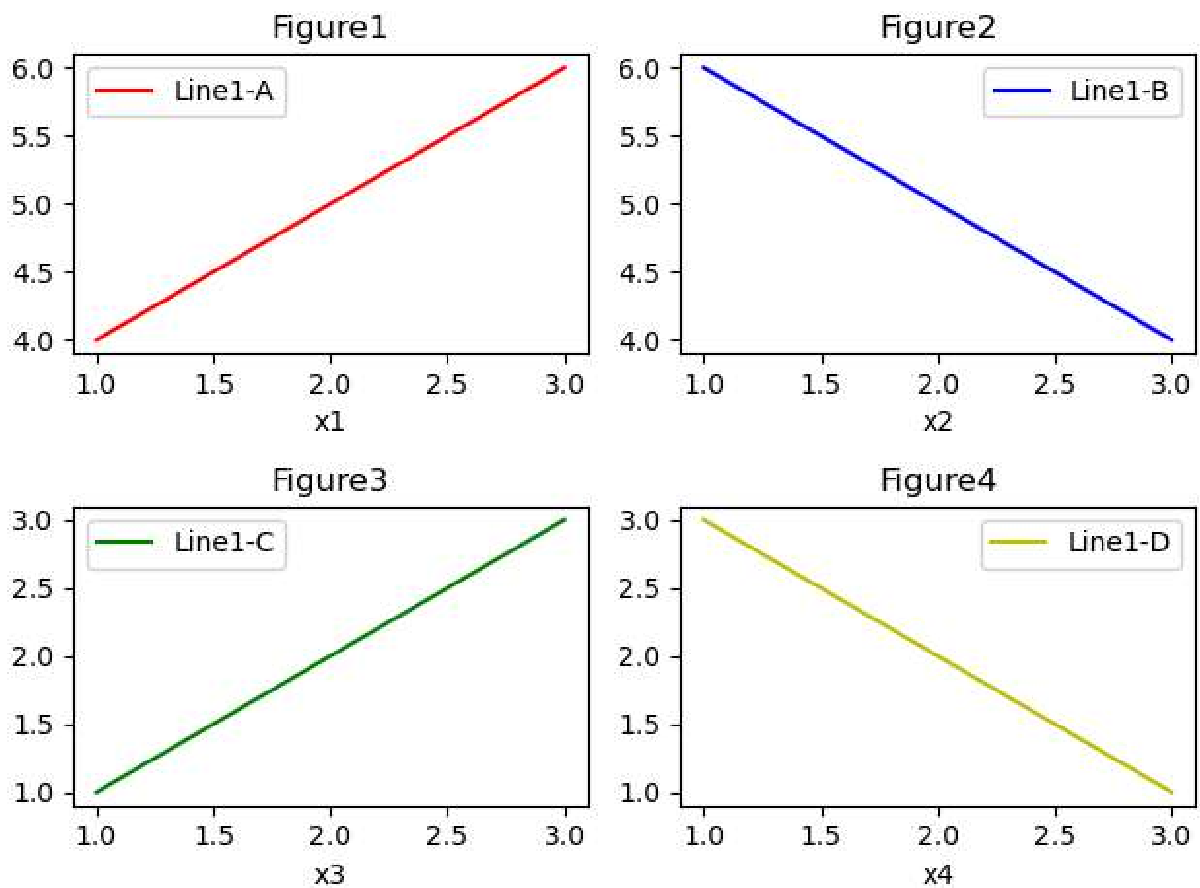
In [38]:
```
plt.show()
```



In [39]:
```
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)

# Axis labels
ax1.set_xlabel('x1')
ax2.set_xlabel('x2')
```

```
ax3.set_xlabel('x3')
ax4.set_xlabel('x4')

# Titles
ax1.set_title('Figure1')
ax2.set_title('Figure2')
ax3.set_title('Figure3')
ax4.set_title('Figure4')

# Add plots with labels
ax1.plot([1,2,3], [4,5,6],'r', label='Line1-A')
ax2.plot([1,2,3], [6,5,4],'b', label='Line1-B')
ax3.plot([1,2,3], [1,2,3],'g', label='Line1-C')
ax4.plot([1,2,3], [3,2,1],'y', label='Line1-D')

# Add legends to each subplot
ax1.legend()
ax2.legend()
ax3.legend()
ax4.legend()

plt.tight_layout()
```

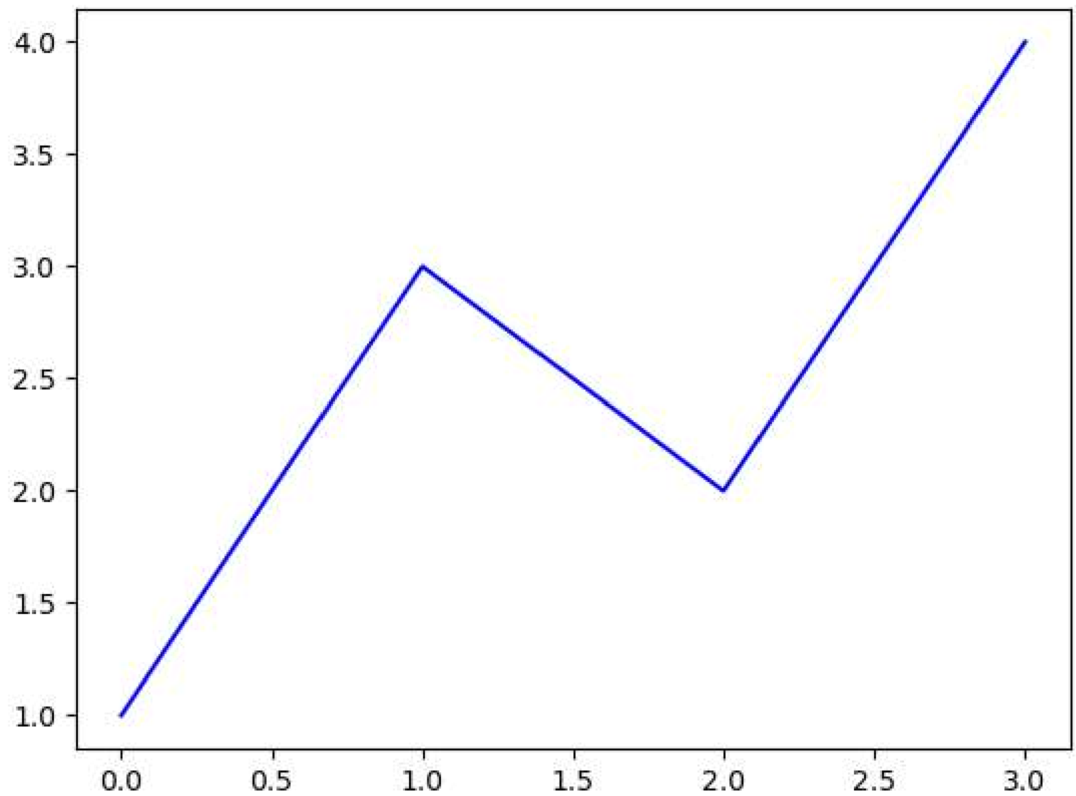
In [40]:
```
plt.show()
```



11. First plot with Matplotlib

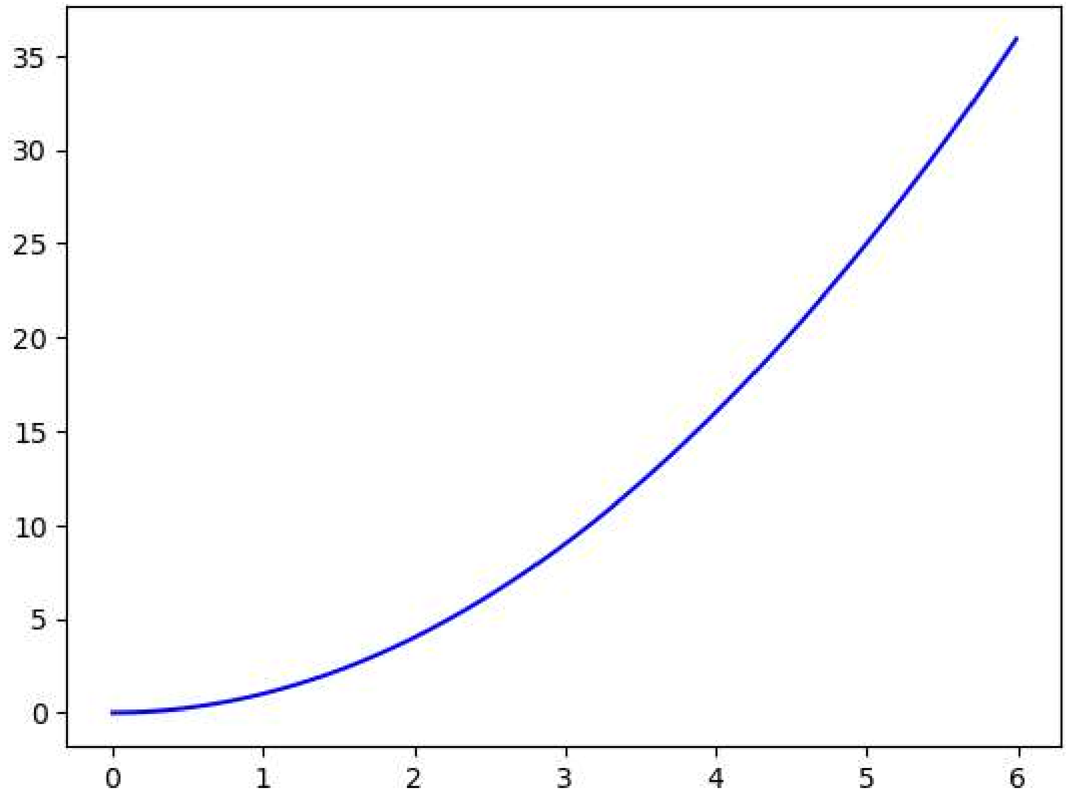
Now, I will start producing plots. Here is the first example:-

In [41]:
```
plt.plot([1,3,2,4],'b-')#X-axis values are automatically assigned as: [0, 1, 2, 3]
plt.figure(figsize=(4,3))#plt.figure() to change the size of plot
plt.show()
```




```
<Figure size 400x300 with 0 Axes>
```

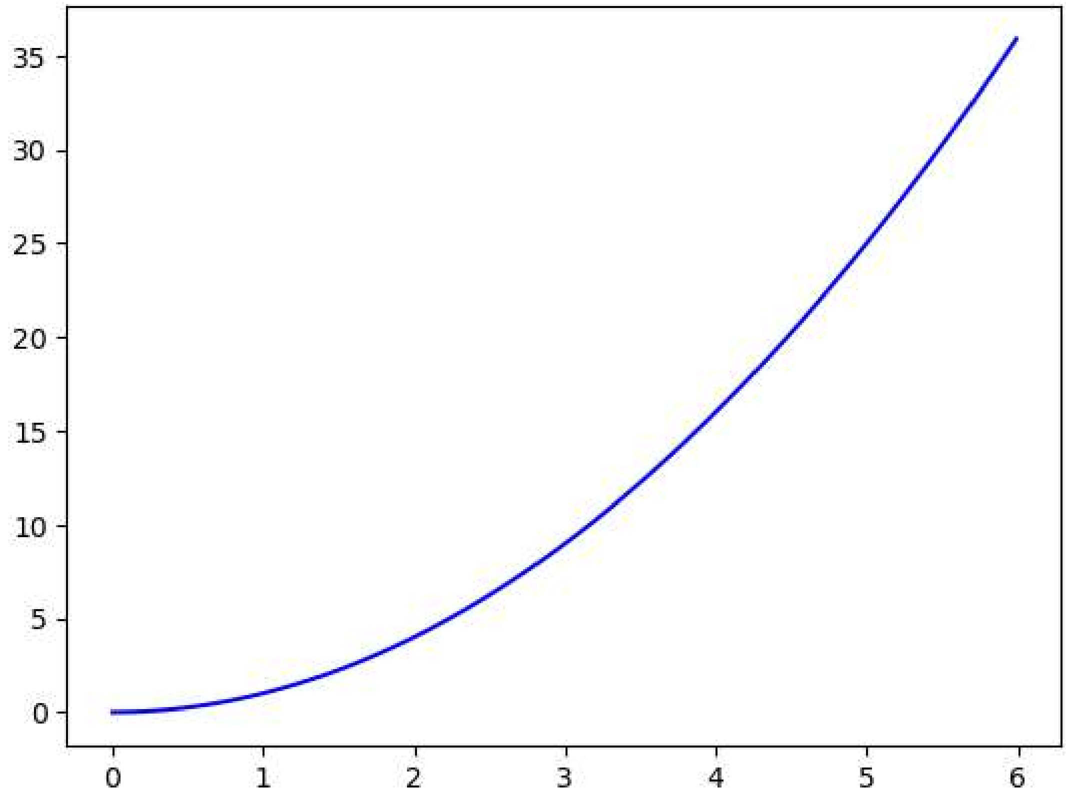
In [42]:
```
x3 = np.arange(0.0,6.0,0.01)#start,stop,step

plt.plot(x3,[xi**2 for xi in x3], 'b-') #xi**2 - Calculate square of each x

plt.show()
```



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).

In [ ]:
```
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()
```

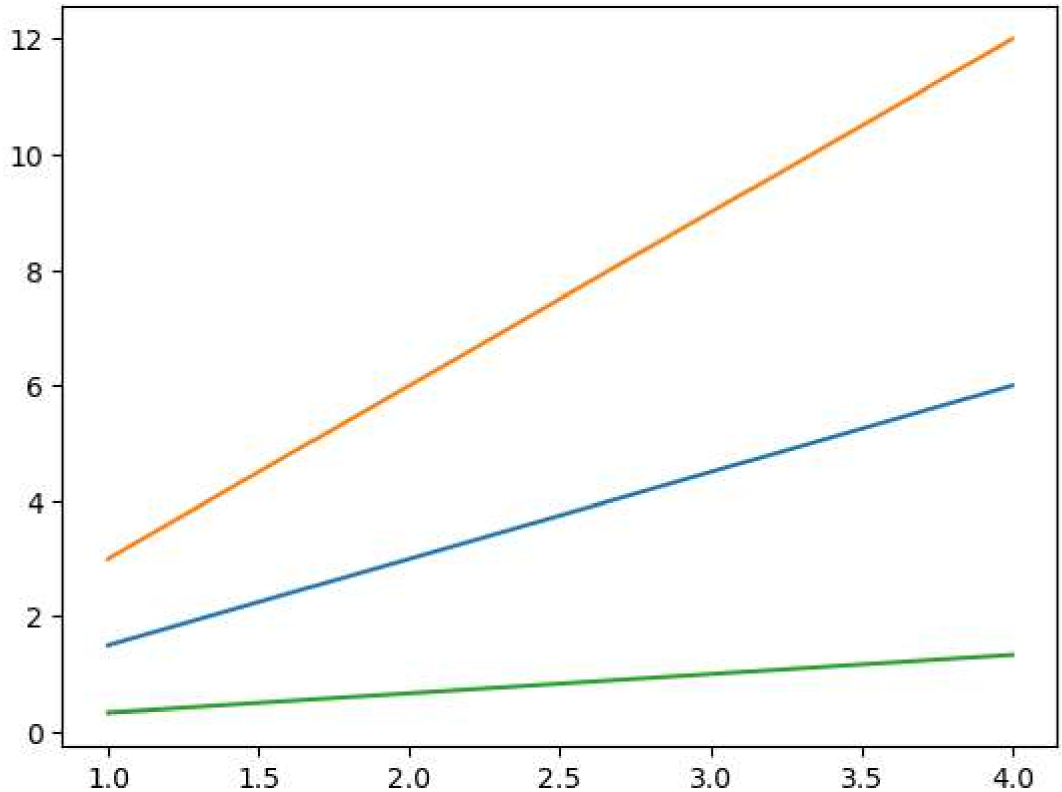
In [43]:
```
#Cleaner and Faster

plt.plot(x3,x3**2, 'b-')
plt.show()
```



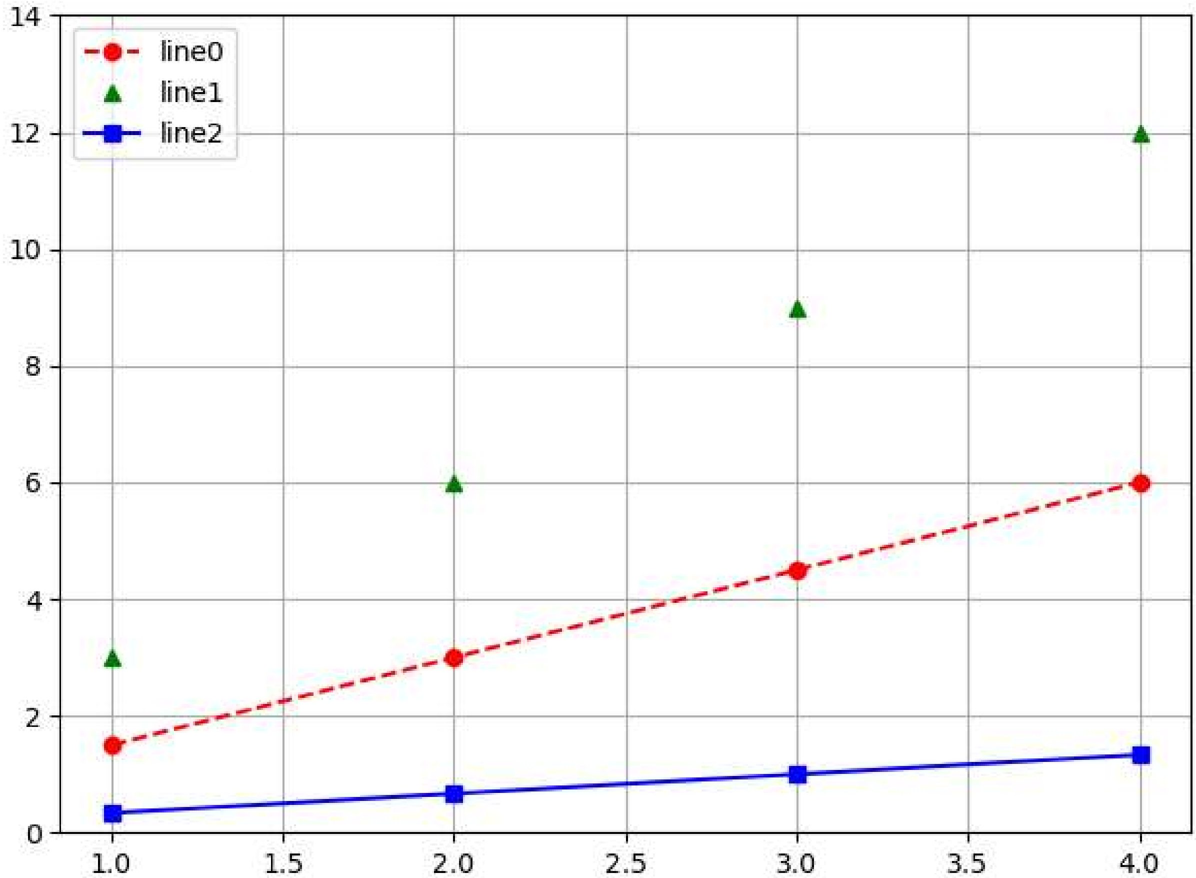
MULTILINE PLOTS

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:-

In [45]:
```
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()
```



In [46]:
```
#Multiline plots (clear)

x4 = range(1,5)

plt.plot(x4,[xi*1.5 for xi in x4],'ro--',label='line0')#o is circle marker to visib

plt.plot(x4,[xi*3 for xi in x4],'g^',label='line1')

plt.plot(x4,[xi/3.0 for xi in x4],'bs-',label='line2')

plt.legend()#Displays a legend box showing the names of all three lines:
            #line0, line1, line2

plt.ylim(0,14)#set y - axis limit

plt.grid(True)#visible y-points

plt.tight_layout()

plt.show()
```



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:-

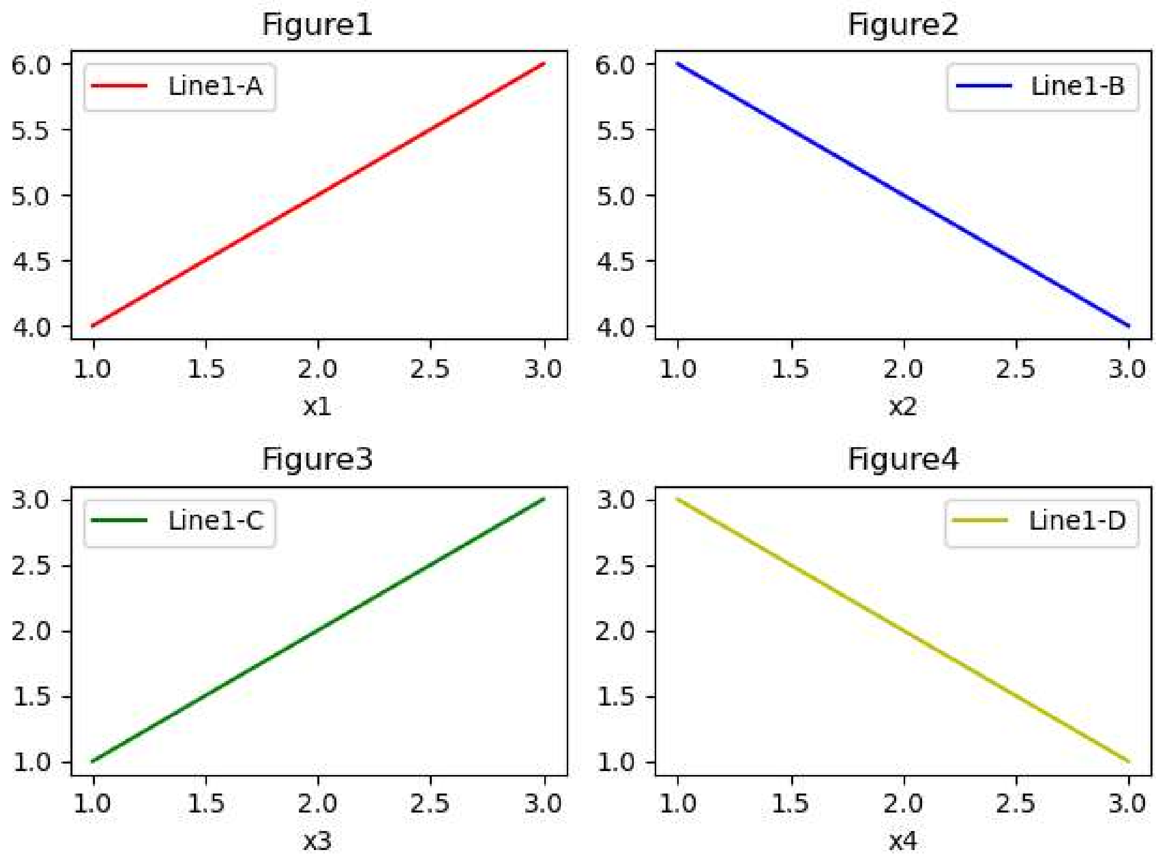
fig.canvas.get_supported_filetypes()

In [47]:

```
### Savings the figure

fig.savefig('plot1.png')
```

In [48]:
```
#Explore the contents of figure

from IPython.display import Image

Image('plot1.png')
```

Out[48]:



In [49]:
```
#Explore support file formats

fig.canvas.get_supported_filetypes()
```

Out[49]:
```
{'eps': 'Encapsulated Postscript',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': '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',
 'webp': 'WebP Image Format'}
```

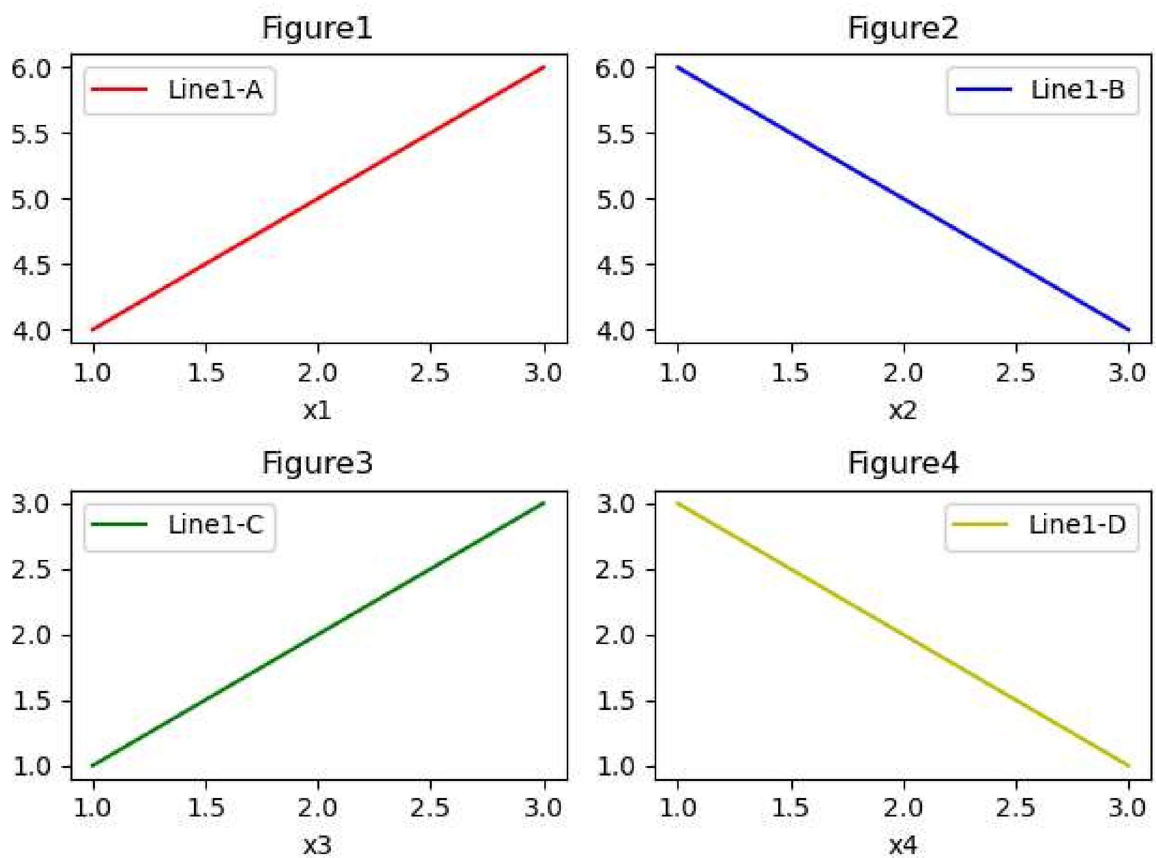
In [50]:
```
fig.savefig("plot.png")
```

In [51]:
```
#This will tell you where to look for the PDF file on your system.

import os
print(os.getcwd())
```

```
C:\Users\akaft
```

In [52]:
```
#This will open and display the PDF file inside the notebook cell.


from IPython.display import Image
Image("plot1.png",width=600,height=400)
```

Out[52]:



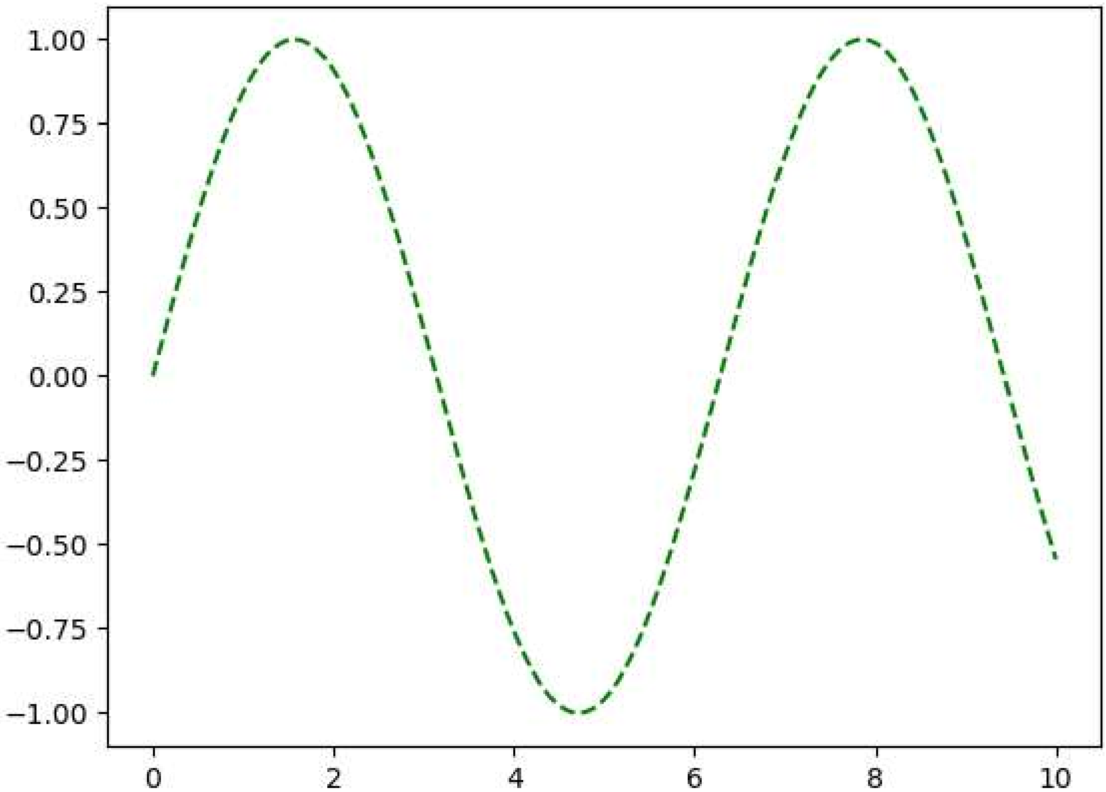
LINE PLOT

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

In [53]:
```
#create figure and axes first

fig = plt.figure()

ax = plt.axes()

#Declare a variable x5
x5= np.linspace(0,10,1000)#Creates 1000 evenly spaced values from 0 to 10
                          #These values are your X-axis data points
```

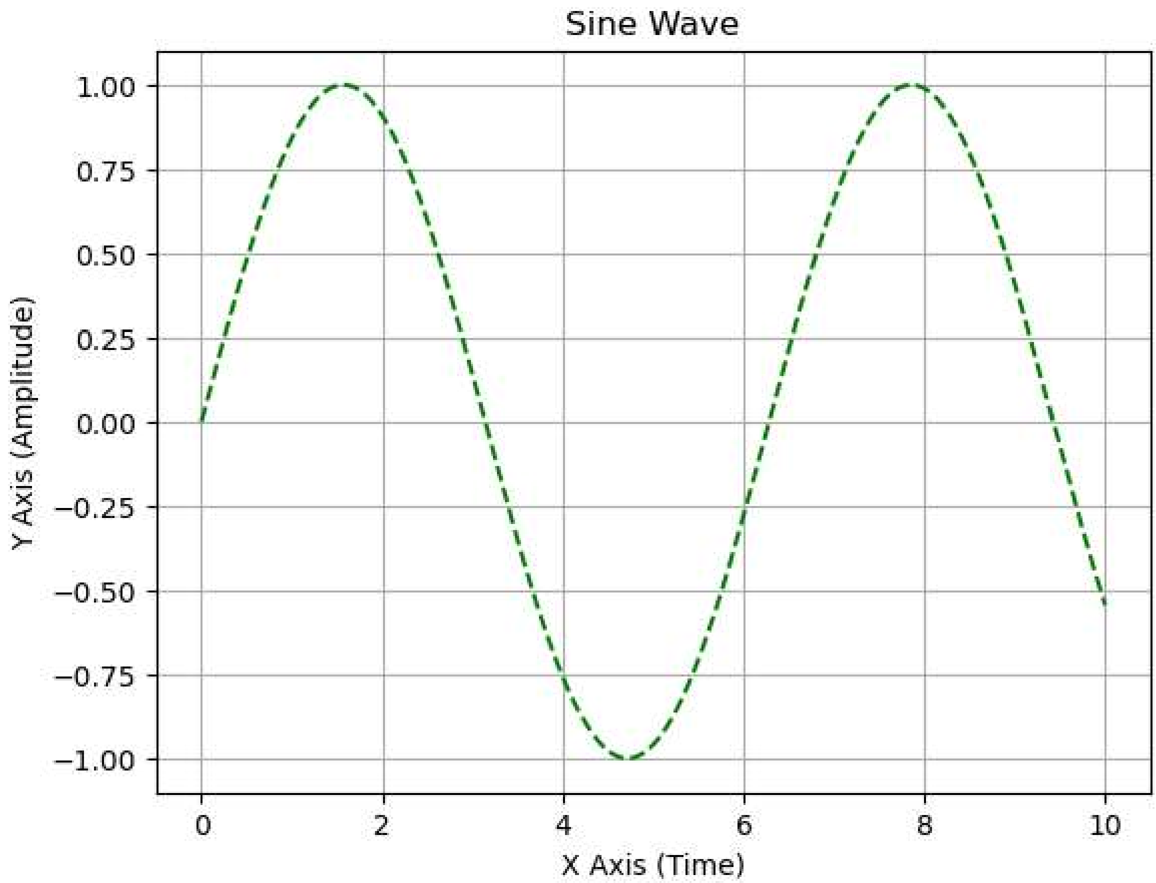
```
#plot the sinusoid function
ax.plot(x5, np.sin (x5), 'g--');#Plots a green dashed line ('g--') for the function
                                #The result is a smooth sine wave
```

In [54]:
```
plt.show()
```



In [55]:
```
#create figure and axes first

fig = plt.figure()

ax = plt.axes()

#Declare a variable x5
x5= np.linspace(0,10,1000)#Creates 1000 evenly spaced values from 0 to 10
                          #These values are your X-axis data points

#plot the sinusoid function
ax.plot(x5, np.sin (x5), 'g--');#Plots a green dashed line ('g--') for the function
                                #The result is a smooth sine wave

#Optional
ax.set_title("Sine Wave")
ax.set_xlabel("X Axis (Time)")
ax.set_ylabel("Y Axis (Amplitude)")
ax.grid(True)
```

In [56]:
```
plt.show()
```



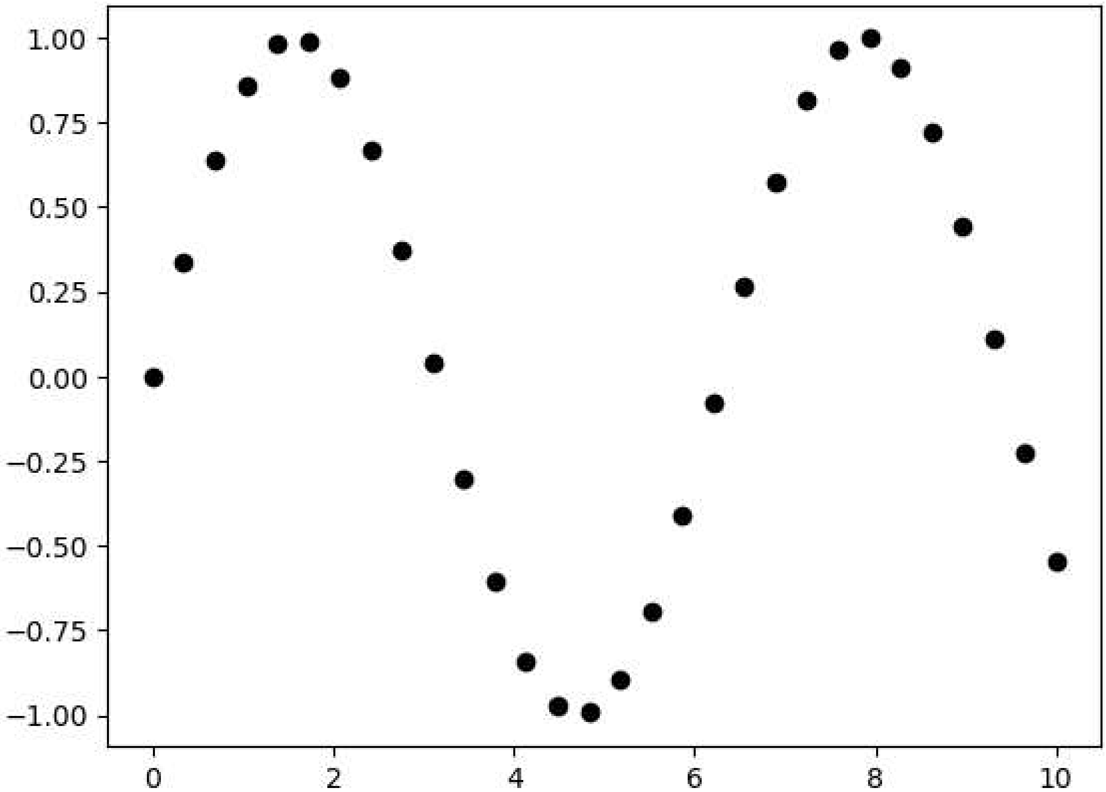
Scatter Plot

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

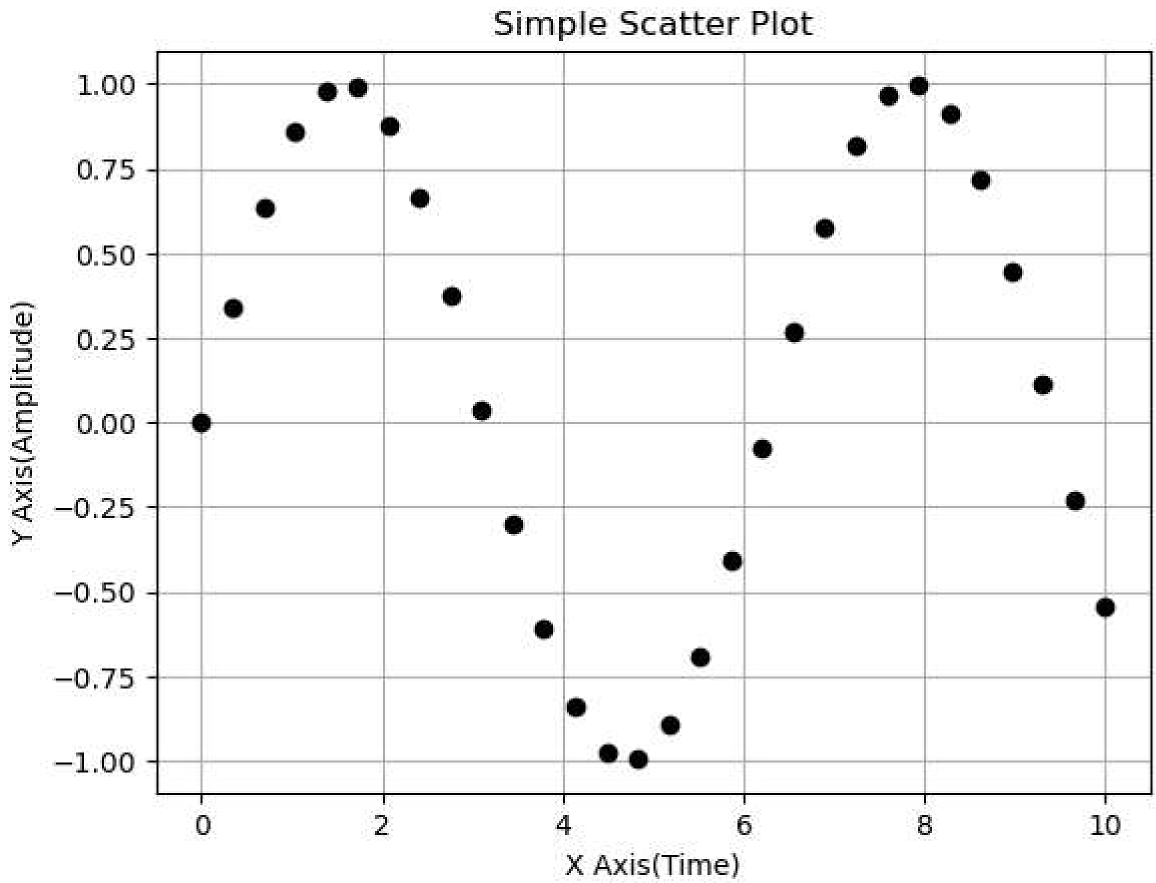
Scatter Plot with plt.plot() We have used plt.plot/ax.plot to produce line plots. We can use the same functions to produce the scatter plots as follows:-

In [58]:
```
x7 = np.linspace(0,10,30)

y7 = np.sin(x7)

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

In [59]:
```
plt.show()
```



In [60]:
```
x7 = np.linspace(0,10,30)

y7 = np.sin(x7)

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

plt.title("Simple Scatter Plot")
plt.xlabel("X Axis(Time)")
plt.ylabel("Y Axis(Amplitude)")
plt.grid(True)
plt.show()
```



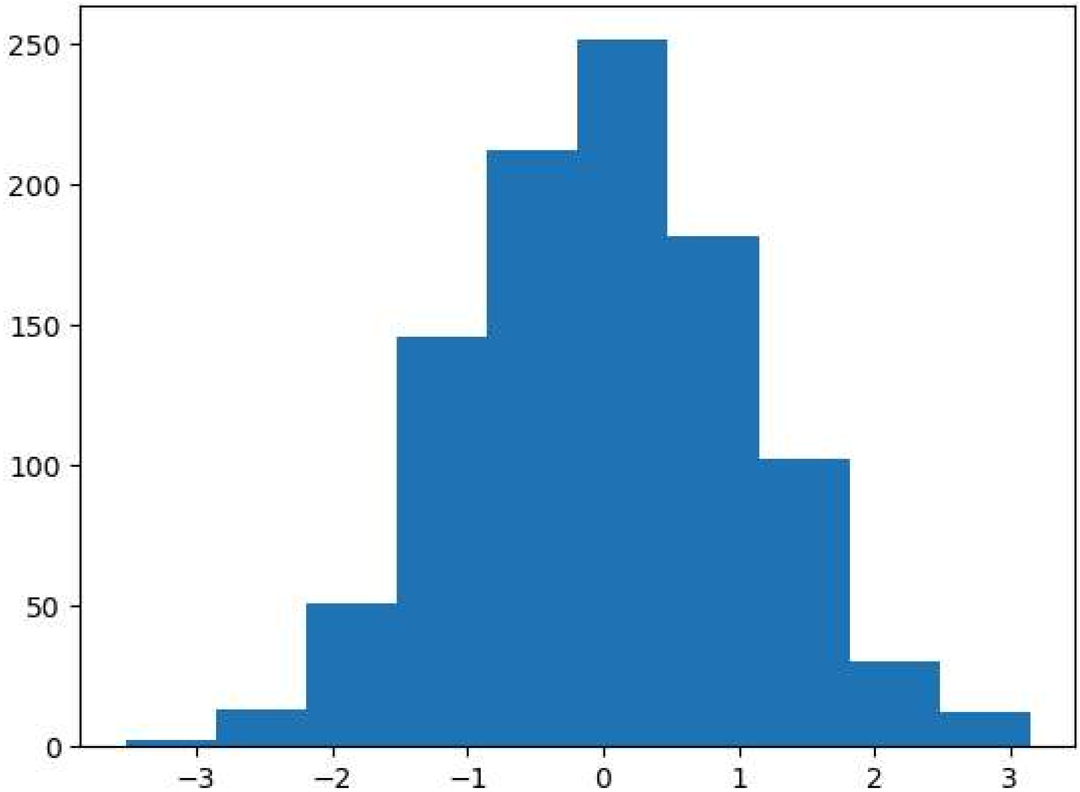
HISTOGRAM

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:-

In [62]:
```
#p-1
data1 = np.random.randn(1000)

plt.hist(data1);
```

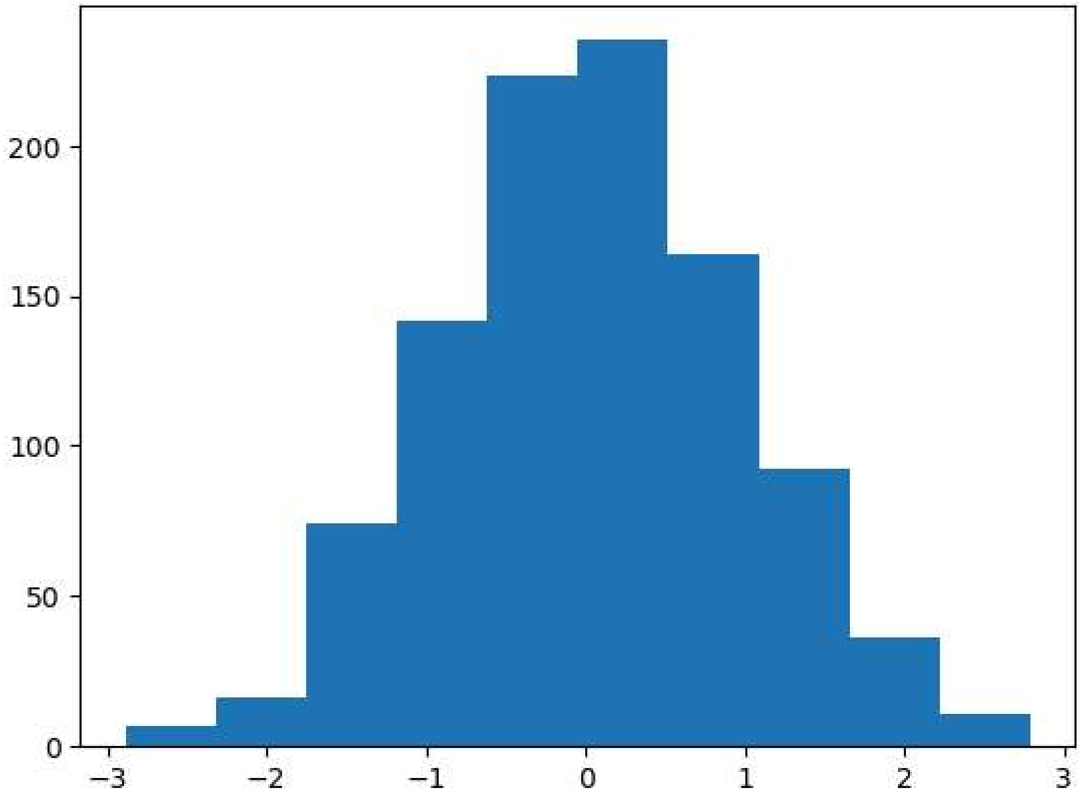
In [63]:
```
plt.show()
```



In [64]:
```
#p-2
np.random.seed(4)
data2 = np.random.randn(1000)

plt.hist(data2)#semicolon not used
```

Out[64]:
```
(array([  7.,  16.,  74., 142., 223., 235., 164.,  92.,  36.,  11.]),
 array([-2.88321807, -2.3162159 , -1.74921374, -1.18221157, -0.6152094 ,
        -0.04820724,  0.51879493,  1.08579709,  1.65279926,  2.21980143,
         2.78680359]),
 <BarContainer object of 10 artists>)
```

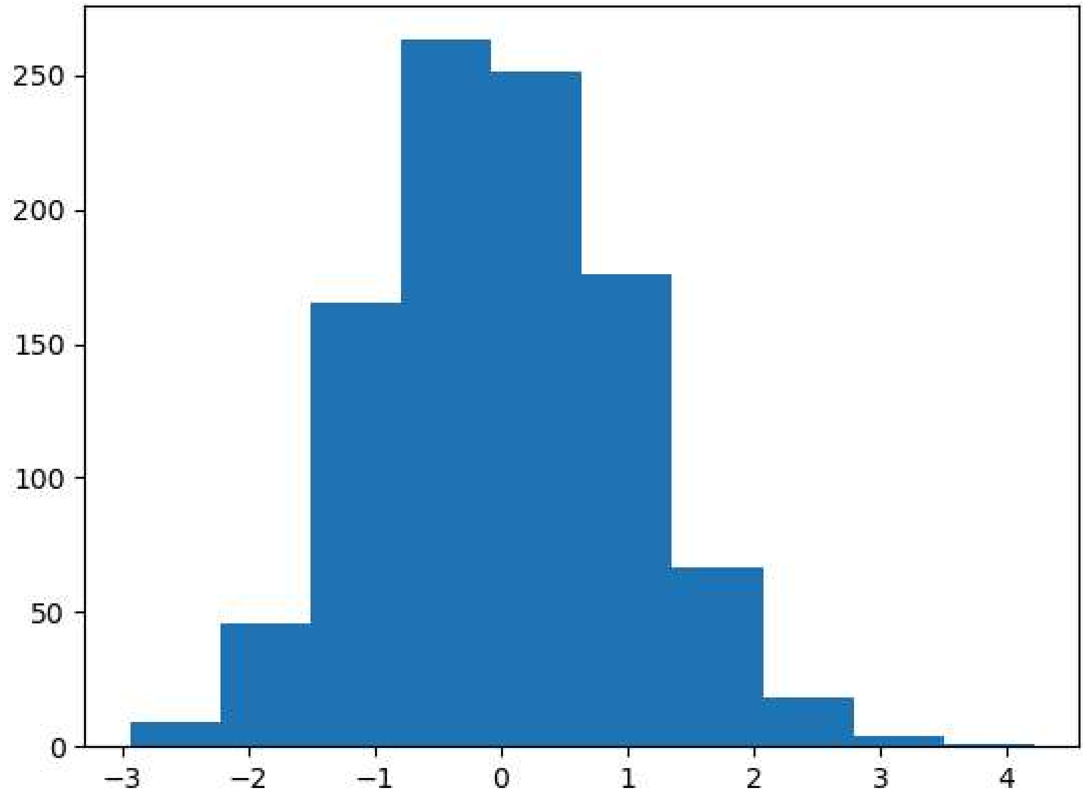
In [65]:
```
plt.show()
```



In [66]:
```
#p-3

#np.random.seed(4)
data2 = np.random.randn(1000)
n= len(data2)

# Using Sturges' Rule to calculate bin
import math

sturges_bins = math.ceil(np.log2(n) + 1)
print("Sturges' Rule Bins:", sturges_bins)

# Using Square Root Rule
sqrt_bins = math.ceil(np.sqrt(n))
print("Square Root Rule Bins:",sqrt_bins)

#rice rule
rice_rule = math.ceil(2 *n**(1/3))
print("Rice Rule:",rice_rule)

plt.hist(data2);#semicolon used
```

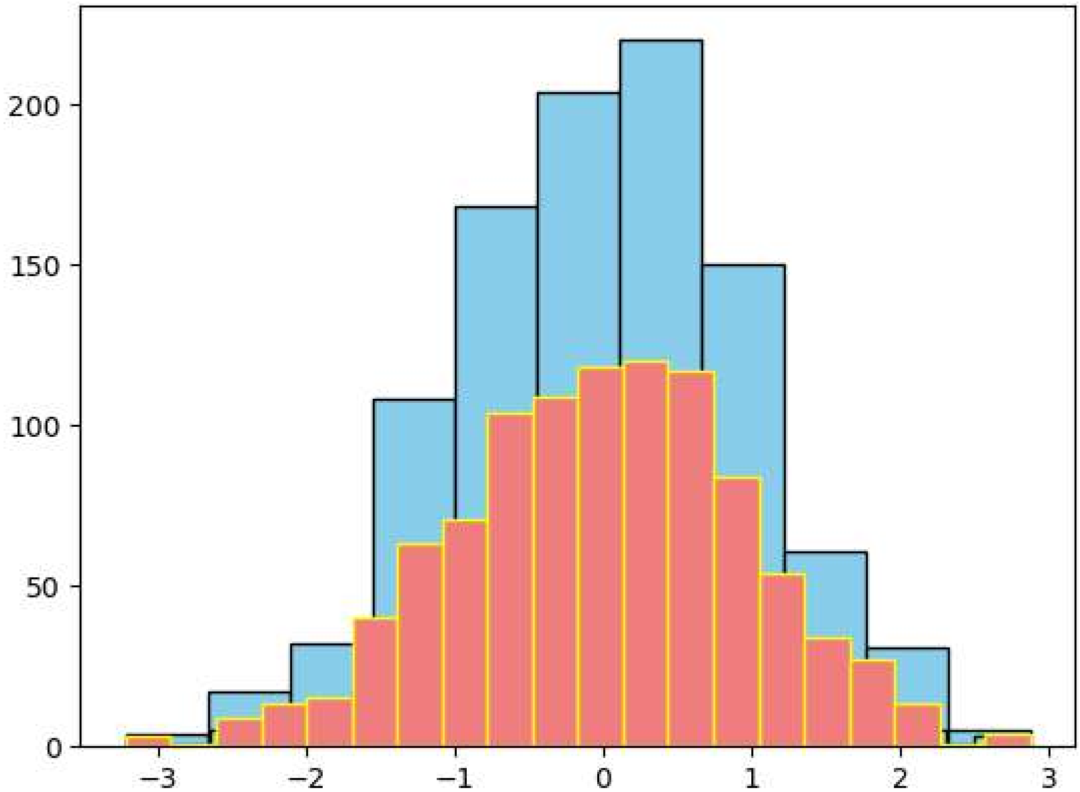
```
Sturges' Rule Bins: 11
Square Root Rule Bins: 32
Rice Rule: 20
```

In [67]:
```
plt.show()
```

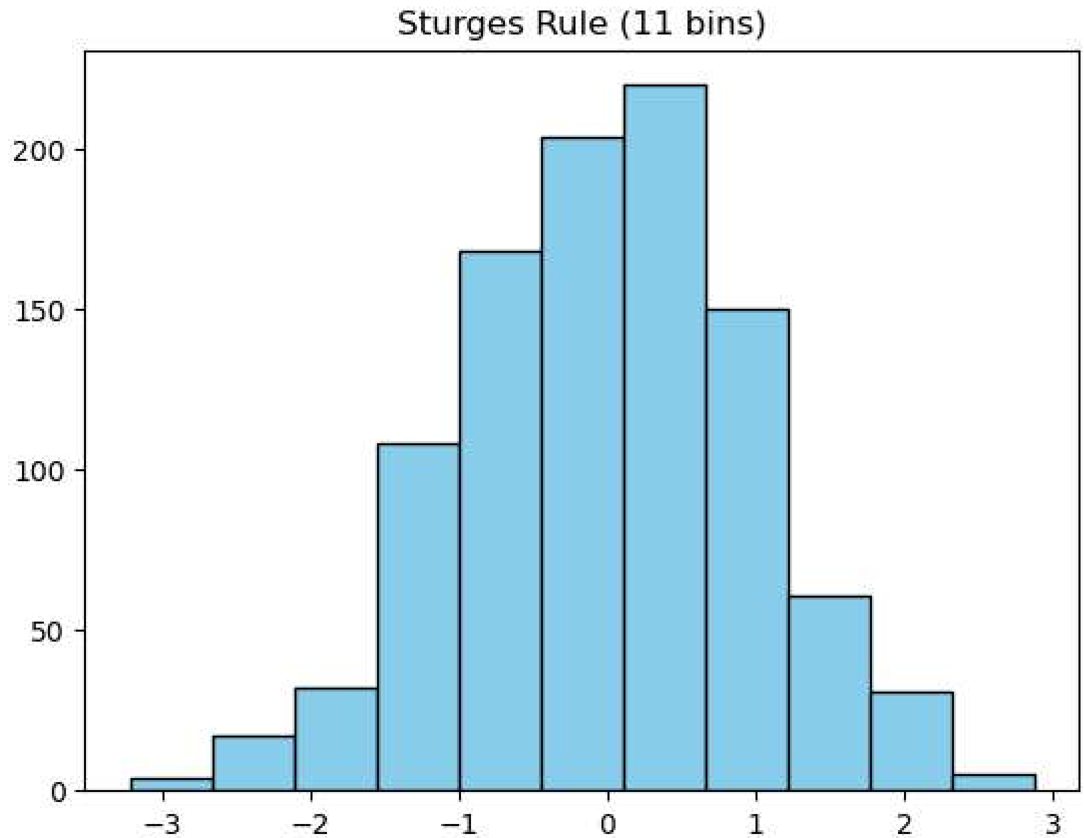


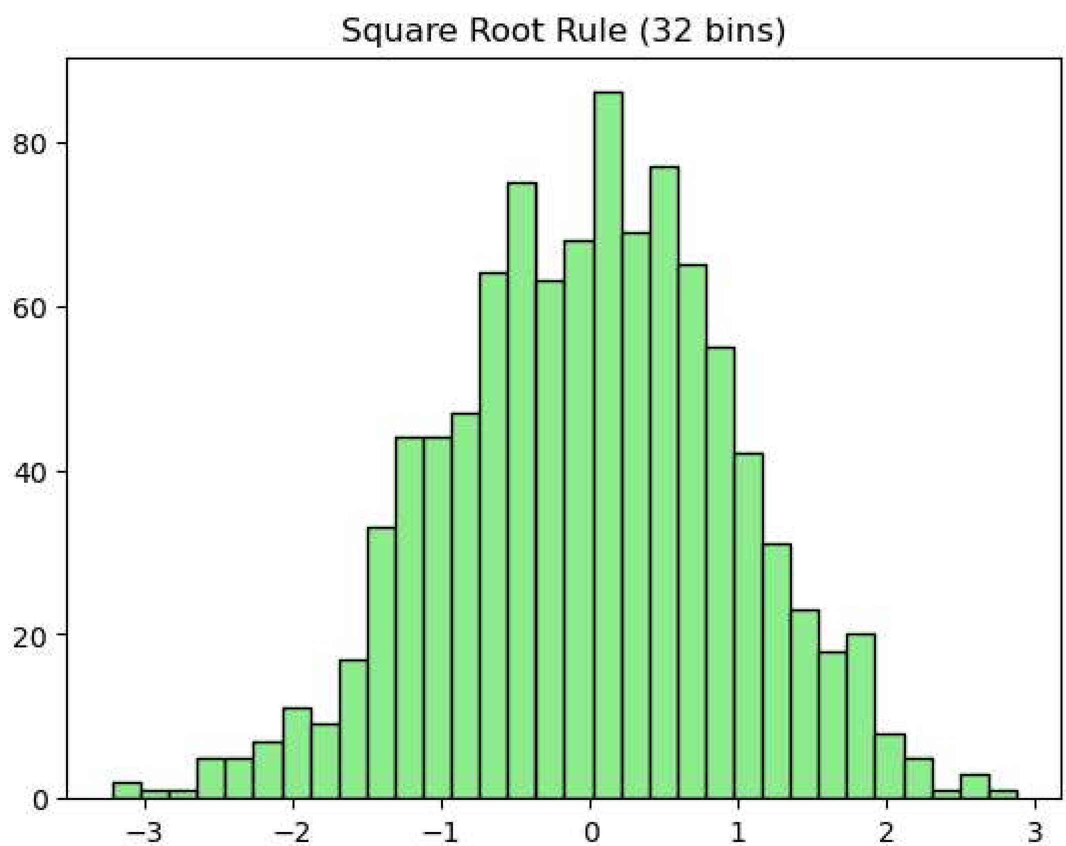
In [68]:
```
#p-4  (overlapped)

#np.random.seed(4)
data2 = np.random.randn(1000)
n= len(data2)

import math

# Using Sturges' Rule to calculate bin
sturges_bins = math.ceil(np.log2(n) + 1)

# Using Square Root Rule
sqrt_bins = math.ceil(np.sqrt(n))

#rice rule
rice_rule = math.ceil(2 *n**(1/3))

#only one bins= parameter is accepted per function call.
plt.hist(data2,bins=sturges_bins,color='skyblue',edgecolor= 'black')
plt.hist(data2,bins=sqrt_bins,color='lightgreen',edgecolor= 'black')
plt.hist(data2,bins=rice_rule,color='lightcoral',edgecolor= 'yellow');
```

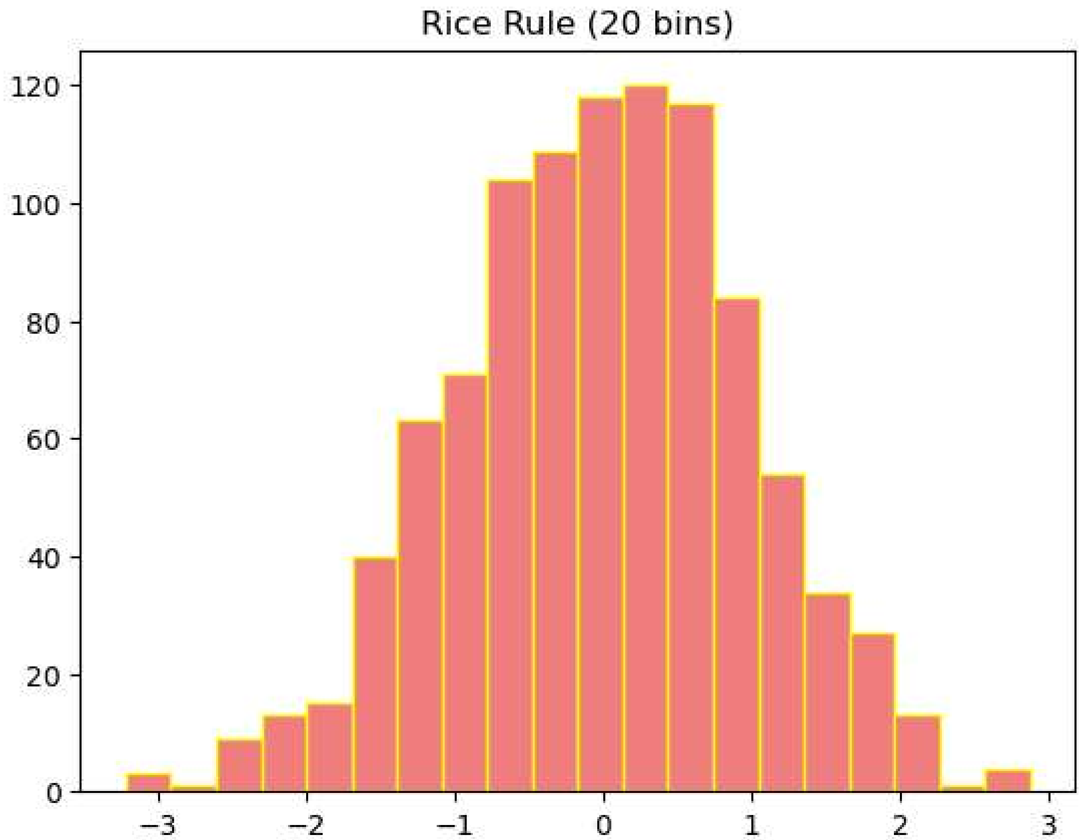
In [69]:
```
plt.show()
```



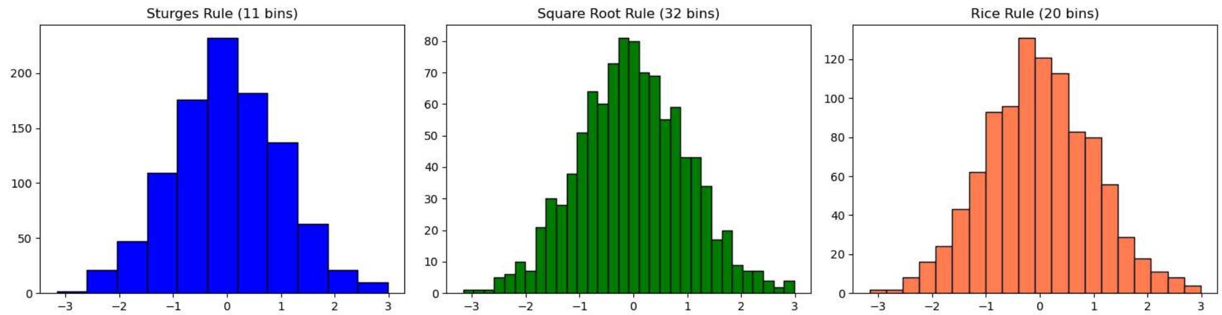
In [70]:
```
#part-5

## ✅ Solution: Use **Separate Plots**
plt.hist(data2, bins=sturges_bins, color='skyblue', edgecolor='black')
plt.title(f"Sturges Rule ({sturges_bins} bins)")
plt.show()

plt.hist(data2, bins=sqrt_bins, color='lightgreen', edgecolor='black')
plt.title(f"Square Root Rule ({sqrt_bins} bins)")
plt.show()

plt.hist(data2, bins=rice_rule, color='lightcoral', edgecolor='yellow')
plt.title(f"Rice Rule ({rice_rule} bins)")
plt.show()
```


Sturges Rule (11 bins)
0
50
100
150
200
−3
−2
−1
0
1
2
3


Square Root Rule (32 bins)
0
20
40
60
80
−3
−2
−1
0
1
2
3



In [71]:
```
#p-6

#import matplotlib.pyplot as plt
#import numpy as np
import math

# Data
data2 = np.random.randn(1000)
n = len(data2)

# Bin rules
sturges_bins = math.ceil(np.log2(n) + 1)
sqrt_bins = math.ceil(np.sqrt(n))
rice_bins = math.ceil(2 * n**(1/3))

# Subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 4))

axs[0].hist(data2, bins=sturges_bins, color='blue', edgecolor='black')
axs[0].set_title(f"Sturges Rule ({sturges_bins} bins)")

axs[1].hist(data2, bins=sqrt_bins, color='green', edgecolor='black')
axs[1].set_title(f"Square Root Rule ({sqrt_bins} bins)")

axs[2].hist(data2, bins=rice_bins, color='coral', edgecolor='black')
axs[2].set_title(f"Rice Rule ({rice_bins} bins)")

plt.tight_layout()
plt.show()
```



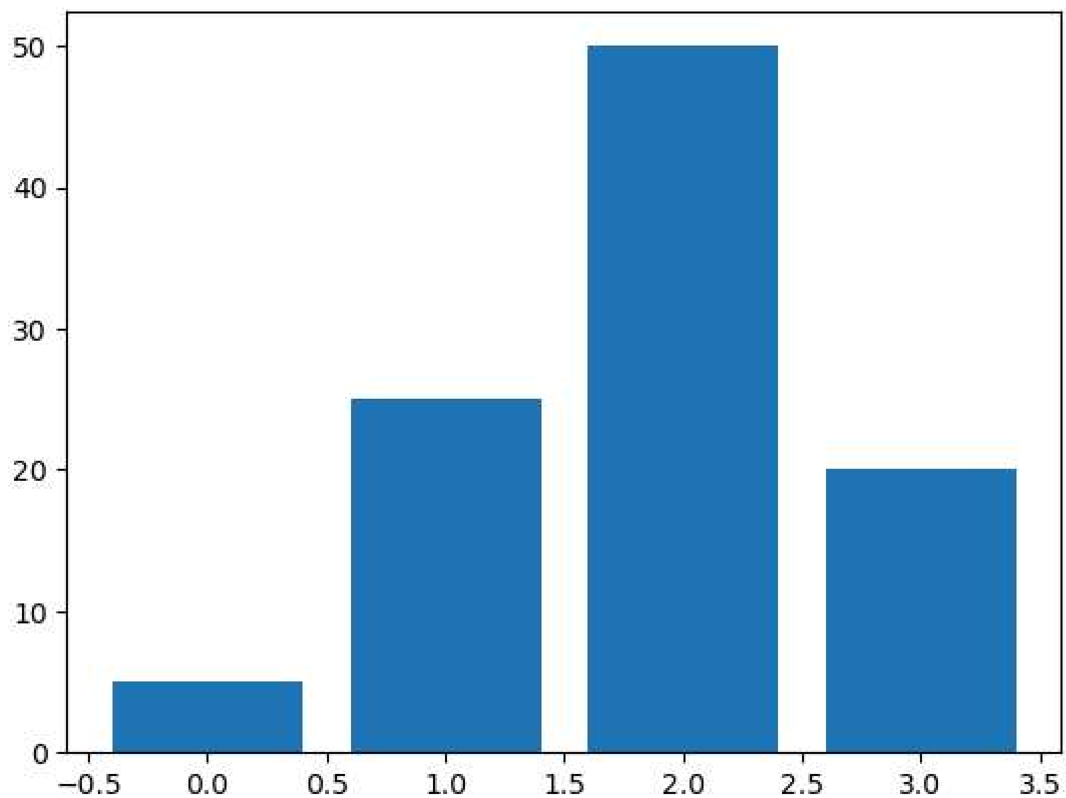
BAR CHART

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:-

In [144…

```
data2 = [5.,25.,50.,20.]#This is your Y-axis data — the height of each bar

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

plt.show()
```

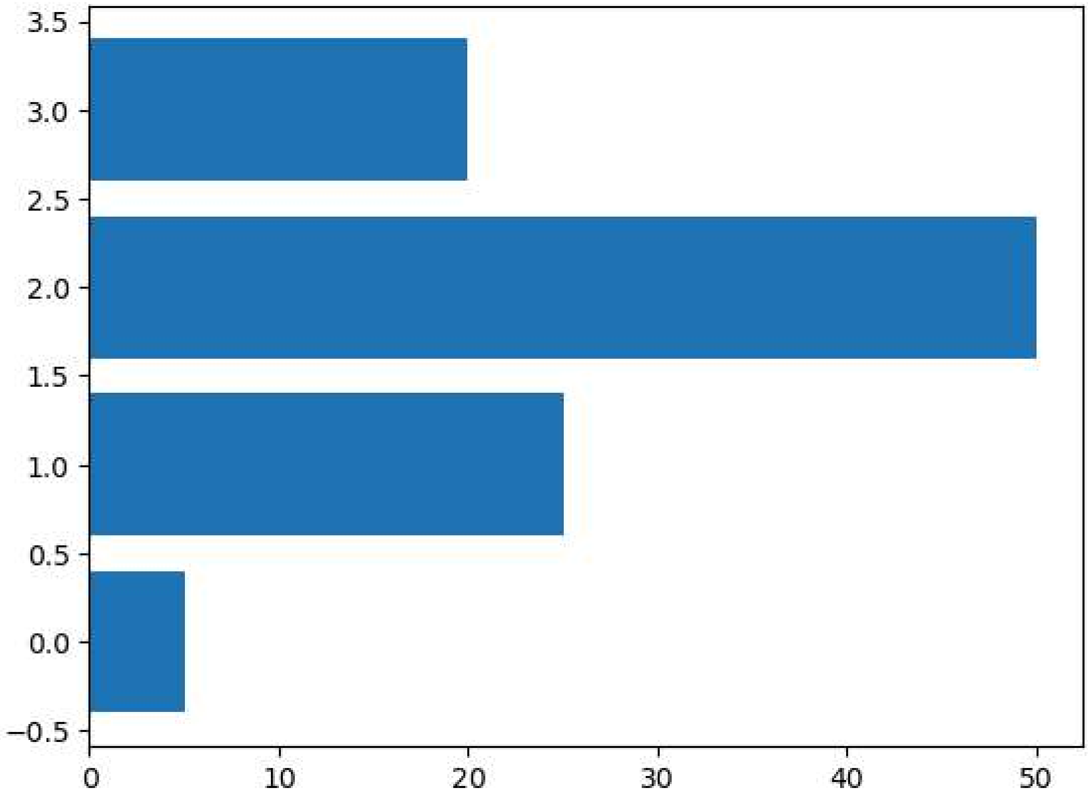


Horizontal Bar Chart

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

In [149…

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

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

plt.show()
```



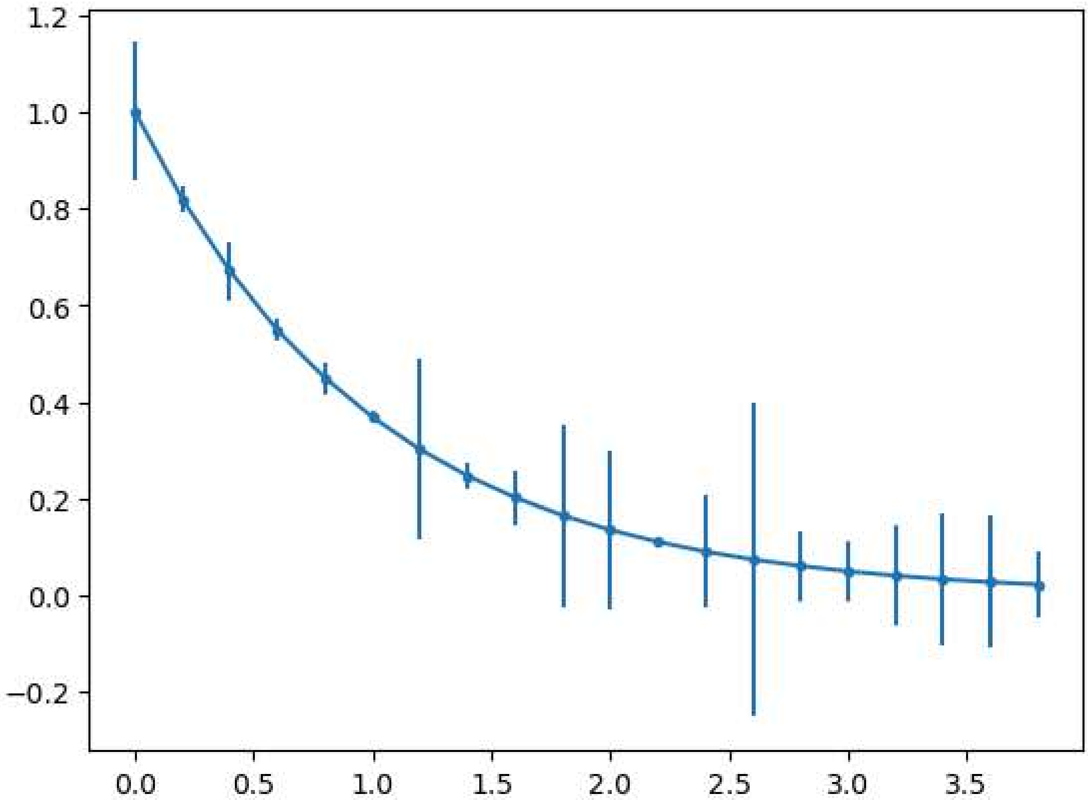
ERROR BAR CHART

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:-

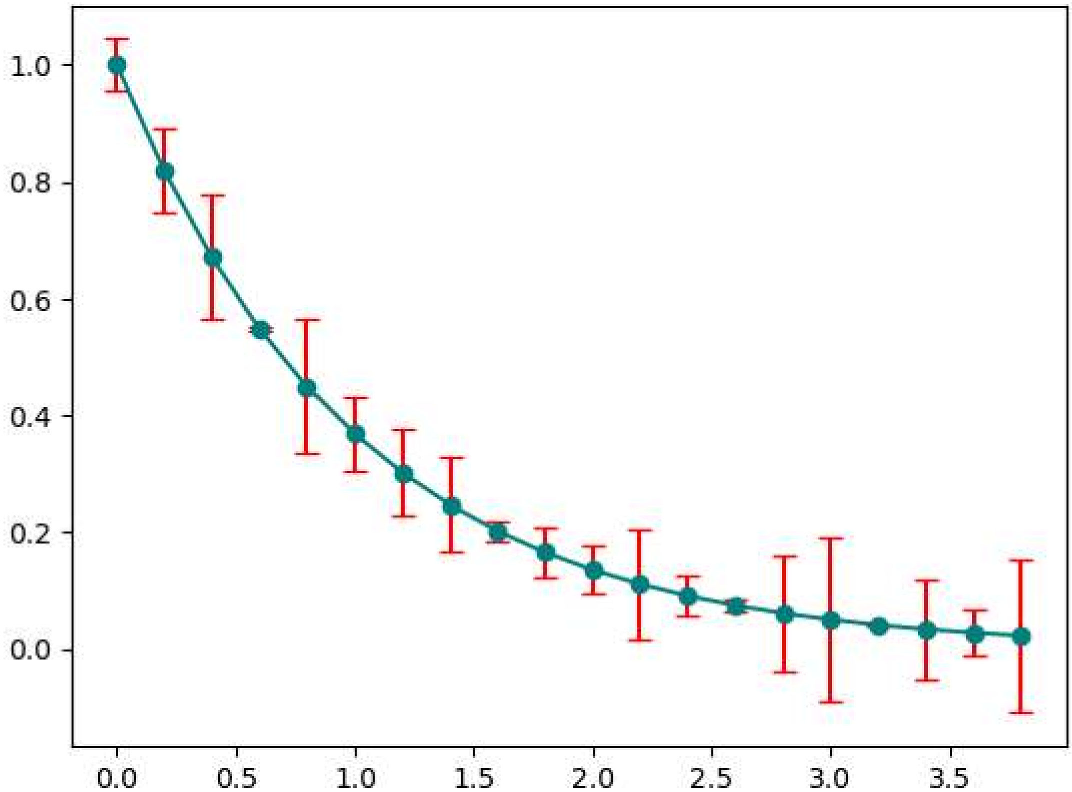
In [152…

```
#p-1

x9 = np.arange(0, 4, 0.2)#start,stop,step

y9 = np.exp(-x9)# Y-axis values: Exponential decay function (y = e^-x)

e1 = 0.1 * np.abs(np.random.randn(len(y9)))# Random error values (positive)

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

plt.show();
```



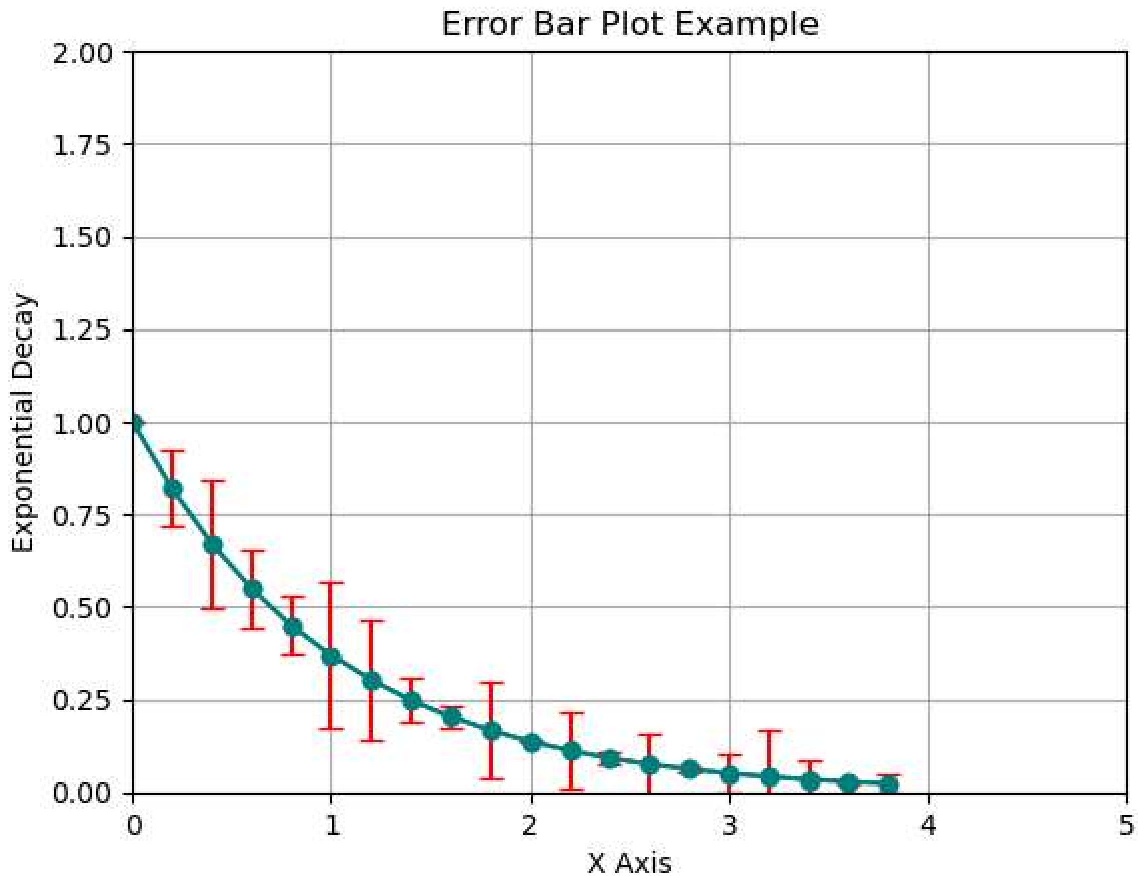
In [158…

```
#p-2

x9 = np.arange(0, 4, 0.2)#start,stop,step

y9 = np.exp(-x9)# Y-axis values: Exponential decay function (y = e^-x)

e1 = 0.1 * np.abs(np.random.randn(len(y9)))# Random error values (positive)

plt.errorbar(x9, y9, yerr = e1, fmt = 'o-',color = "teal",ecolor='red',capsize=4)

plt.show();
```



In [178…

```
#P-3

x9 = np.arange(0, 4, 0.2)#start,stop,step

y9 = np.exp(-x9)# Y-axis values: Exponential decay function (y = e^-x)-Exponential
                #np.exp() means e to the power of (Euler’s number ≈ 2.718).

#this line used to cret error values,also called uncertainty or variability.
e1 = 0.1 * np.abs(np.random.randn(len(y9)))# random.randn(cret list of randome num
#o.1- To make error bars realistic, small, and clear #np.abs-absolute value-> it re
#You can change it to whatever you want:(0.05 -small error bar/0.2,0.5 larger error

plt.errorbar(x9, y9, yerr = e1, fmt = 'o-',color = "teal",ecolor='red',capsize=4)
plt.ylim(0,2)
plt.xlim(0,5)
plt.title("Error Bar Plot Example")
plt.xlabel("X Axis")
plt.ylabel("Exponential Decay")
plt.grid(True)
plt.show()
```

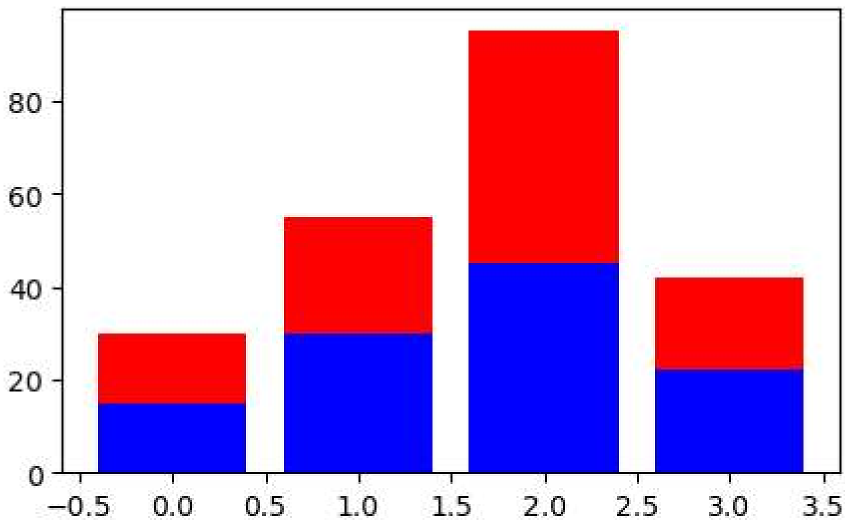


STACKED BAR CHART

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

In [195…

```
#use of bottom=A make this stacked bar chat

plt.figure(figsize=(5,3))#width = 5 inches, height = 3 inches

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)#bottom=A means: start the red bars
                                      #from the top of the blue bars.
plt.show()
```

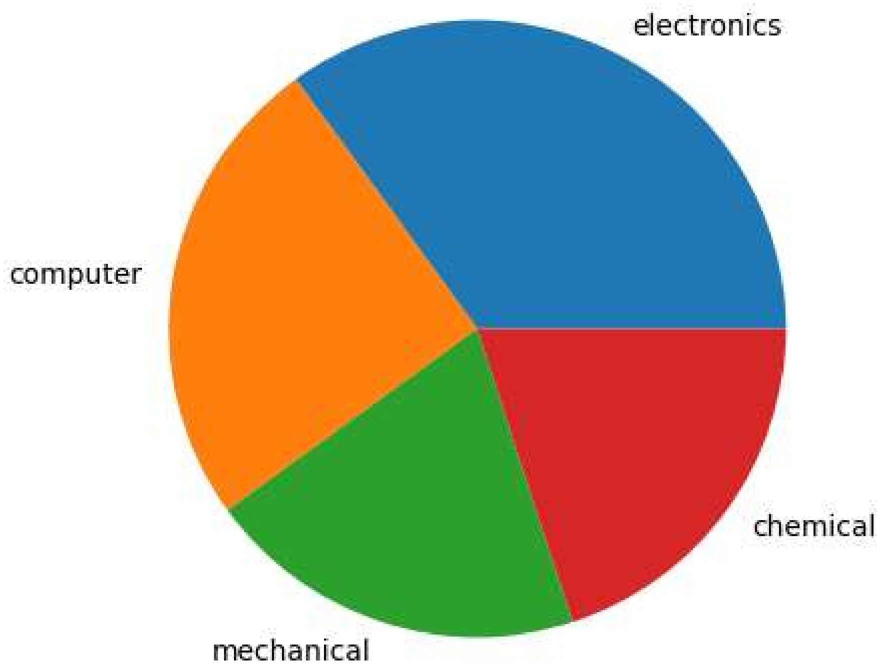


The optional bottom parameter of the plt.bar() function allows us to specify a starting position for a bar. Instead of running from zero to a value, it will go from the bottom to value. The first call to plt.bar() plots the blue bars. The second call to plt.bar() plots the red bars, with the bottom of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

Pie Chart

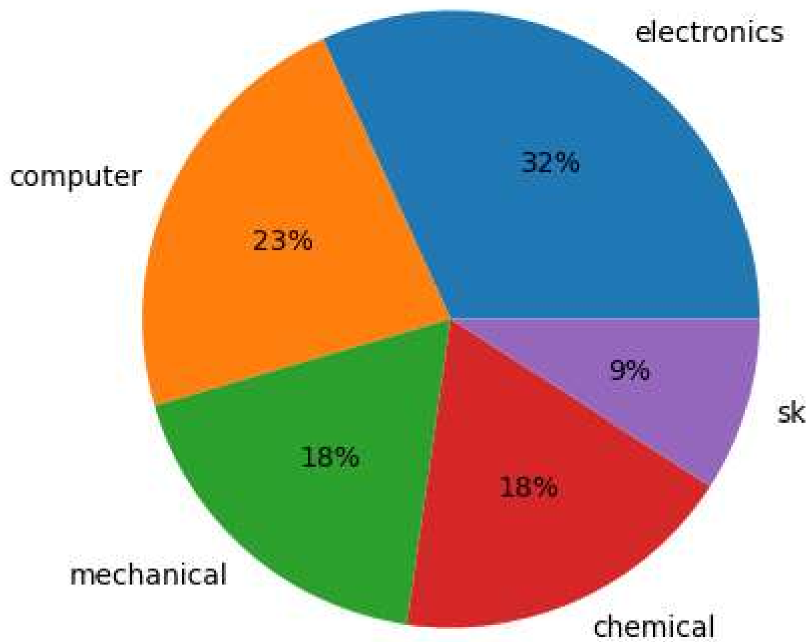
In [216…

```
#p-1

plt.figure (figsize=(5,5))#A square size is good for pie charts,
                            #so the circle looks neat and not stretched.
x10 = [35,25,20,20]

labels = ['electronics', 'computer', 'mechanical','chemical']

plt.pie(x10,labels=labels);

plt.show()
```



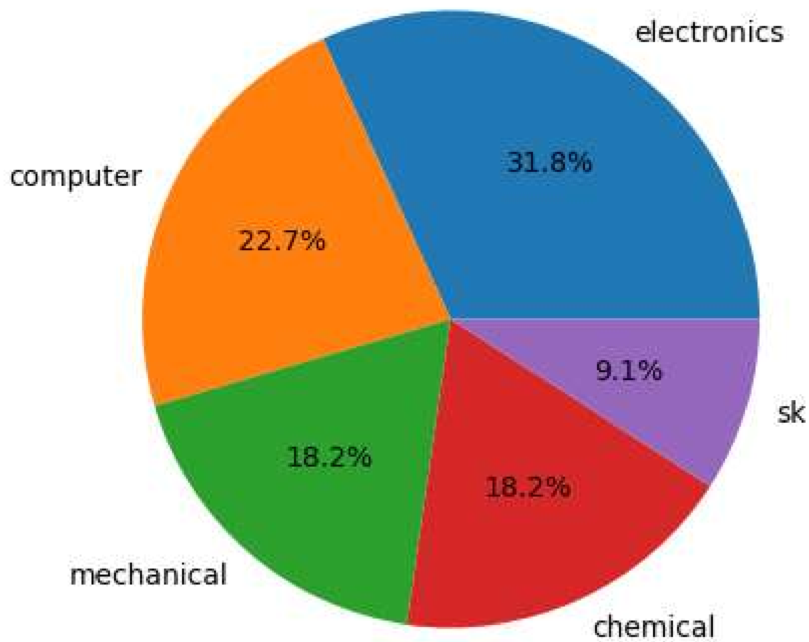
In [222…

```
#P-2

plt.figure (figsize=(5,5))#A square size is good for pie charts,
                          #so the circle looks neat and not stretched.
x10 = [35,25,20,20,10]

labels = ['electronics', 'computer', 'mechanical','chemical','sk']

plt.pie(x10,labels=labels,autopct='%1.0f%%');#autopct='%1.1f%%'- add % labels on th

plt.show()
```



In [224…

```
#p-3

plt.figure (figsize=(5,5))#A square size is good for pie charts,
                         #so the circle looks neat and not stretched.
x10 = [35,25,20,20,10]

labels = ['electronics', 'computer', 'mechanical','chemical','sk']

plt.pie(x10,labels=labels,autopct='%1.1f%%');#autopct='%1.1f%%'- add % labels on th

plt.show()
```



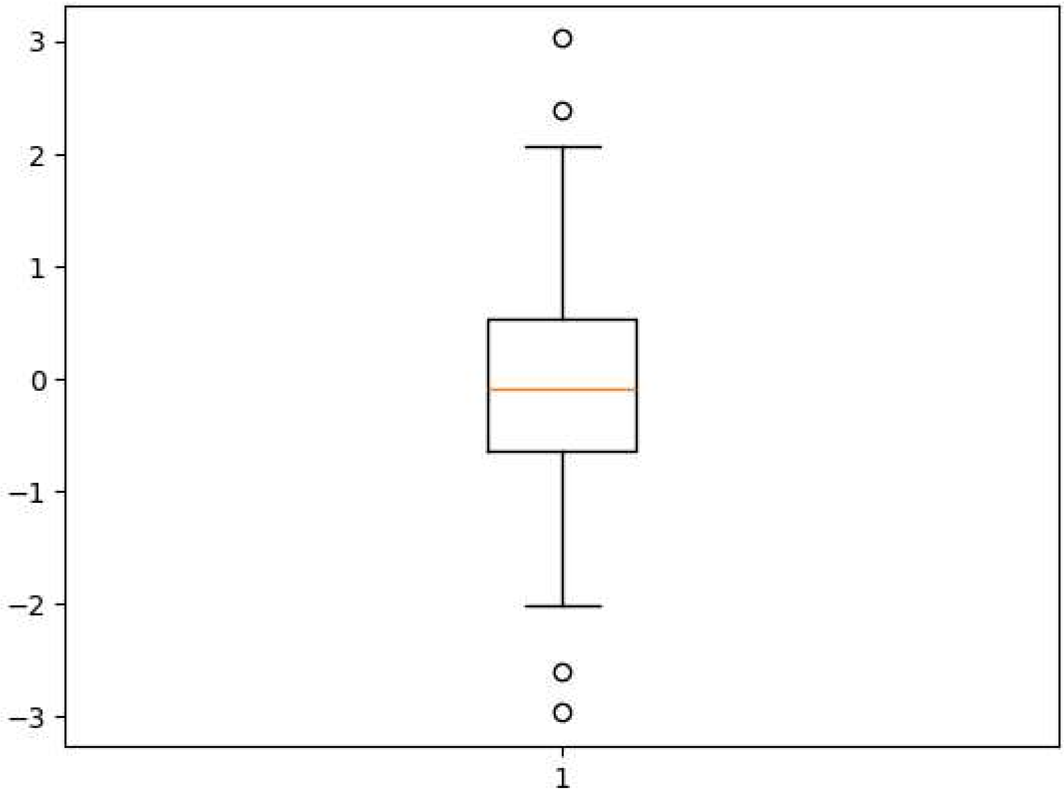
BOX PLOT

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:-

In [229…

```
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.

📦 What does a boxplot show?

Box in the center → the middle 50% of the data (called interquartile range).

Line inside the box → the median (middle value).

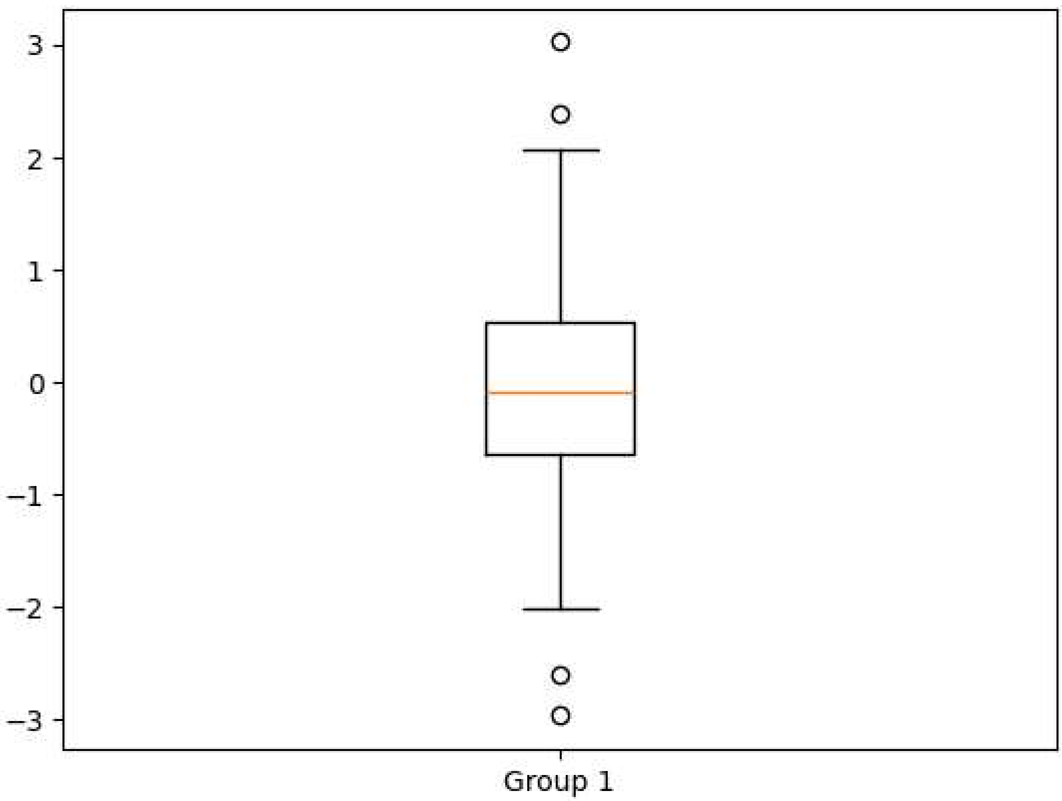
Whiskers (lines extending outside) → show how far the rest of the data spreads.

Circles outside the whiskers → these are called outliers.

In [231…

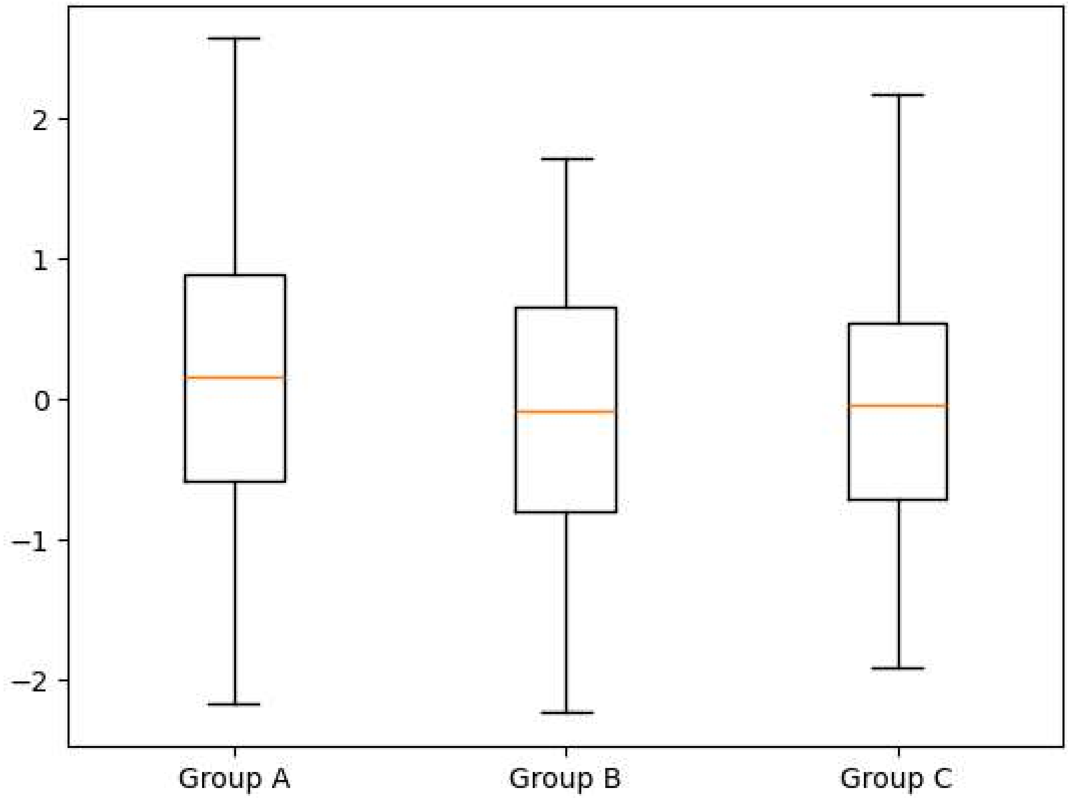
```
#P-2(Manually set X-axis ticks and labels:)
#single boxplot
plt.boxplot(data3)
```

```
plt.xticks([1], ['Group 1'])  # X-axis tick at position 1, labeled 'Group 1'
plt.show()
```



In [239…

```
#P-3 (Multiple Datasets-xaxis appears clearly

data = [np.random.randn(100), np.random.randn(100), np.random.randn(100)]
plt.boxplot(data)
plt.xticks([1, 2, 3], ['Group A', 'Group B', 'Group C'])
plt.show()
```



AREA CHART

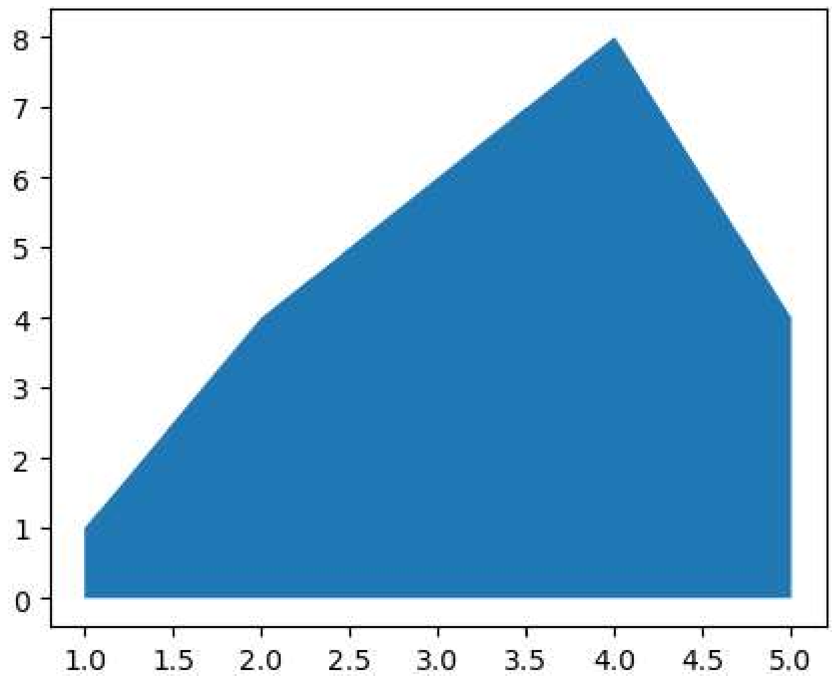
I have created a basic Area chart. I could also use the stackplot function to create the Area chart as follows:-

plt.stackplot(x12, y12)

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

In [248…

```
#P-1
plt.figure(figsize=(5,4))

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

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



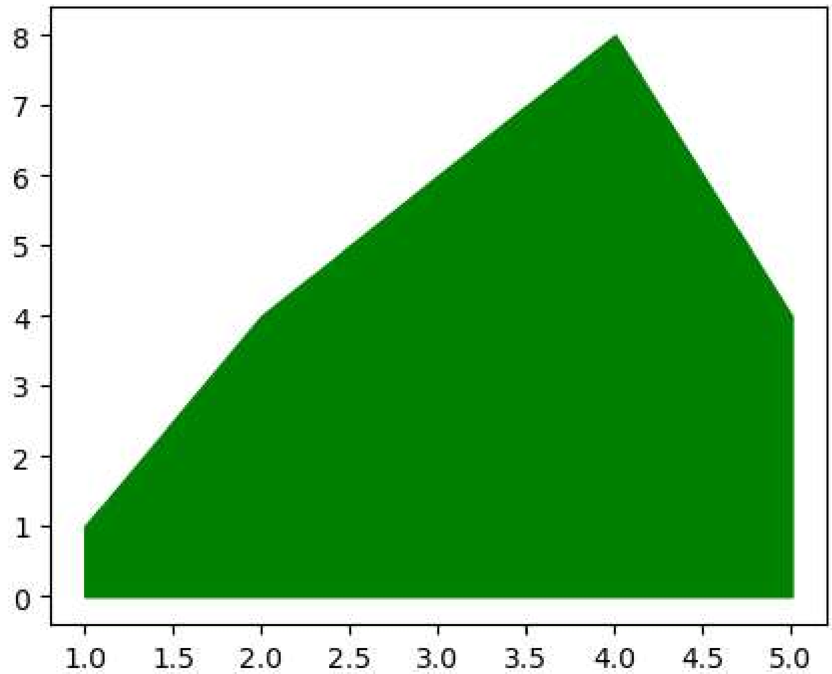
I have created a basic Area chart. I could also use the stackplot function to create the Area chart as follows:-

plt.stackplot(x12, y12)

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

In [258…

```
#P-2

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

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

#Area plot
plt.fill_between(x12,y12,color= 'green', alpha= 0.6)#transparency level (0 = transp
plt.show()                                          #its range between 0-1
```



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.

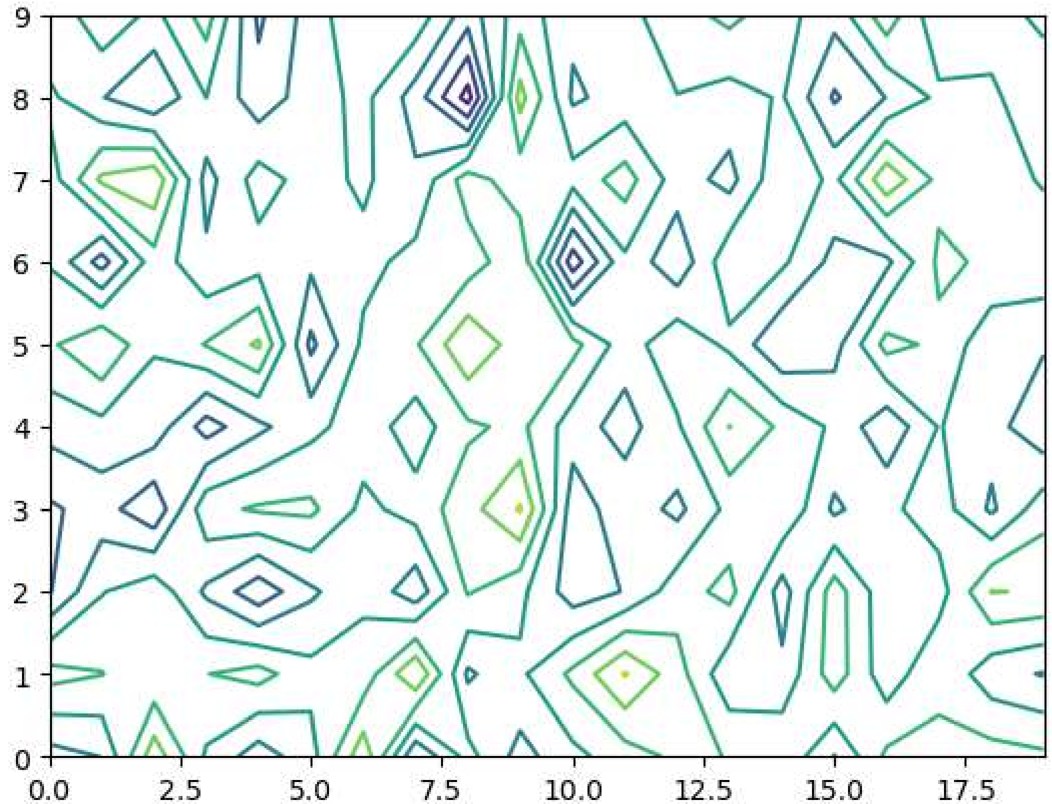
There are lot of applications of Contour lines in several fields such as meteorology(for temperature, pressure, rain, wind speed), geography, magnetism, engineering, social sciences and so on.

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:-

In [263…

```
#create a matrix

matrix1 = np.random.randn(10,20)

cp = plt.contour(matrix1)

plt.show()
```



Styles with Matplotlib Plots

The Matplotlib version 1.4 which was released in August 2014 added a very convenient style module. It includes a number of new default stylesheets, as well as the ability to create and package own styles.

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

print(plt.style.availabe)

In [266…
```
#view list of all available styles

print(plt.style.available)
```

```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'b
mh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale',
'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-colorblind', 'seaborn-v0_8-dar
k', 'seaborn-v0_8-dark-palette', 'seaborn-v0_8-darkgrid', 'seaborn-v0_8-deep', 'seab
orn-v0_8-muted', 'seaborn-v0_8-notebook', 'seaborn-v0_8-paper', 'seaborn-v0_8-paste
l', 'seaborn-v0_8-poster', 'seaborn-v0_8-talk', 'seaborn-v0_8-ticks', 'seaborn-v0_8-
white', 'seaborn-v0_8-whitegrid', 'tableau-colorblind10']
```

In [ ]:
```
#P-1
#set styles for plots
plt.style.use('seaborn-v0_8-bright')
```

In [115…
```
#P-2
```

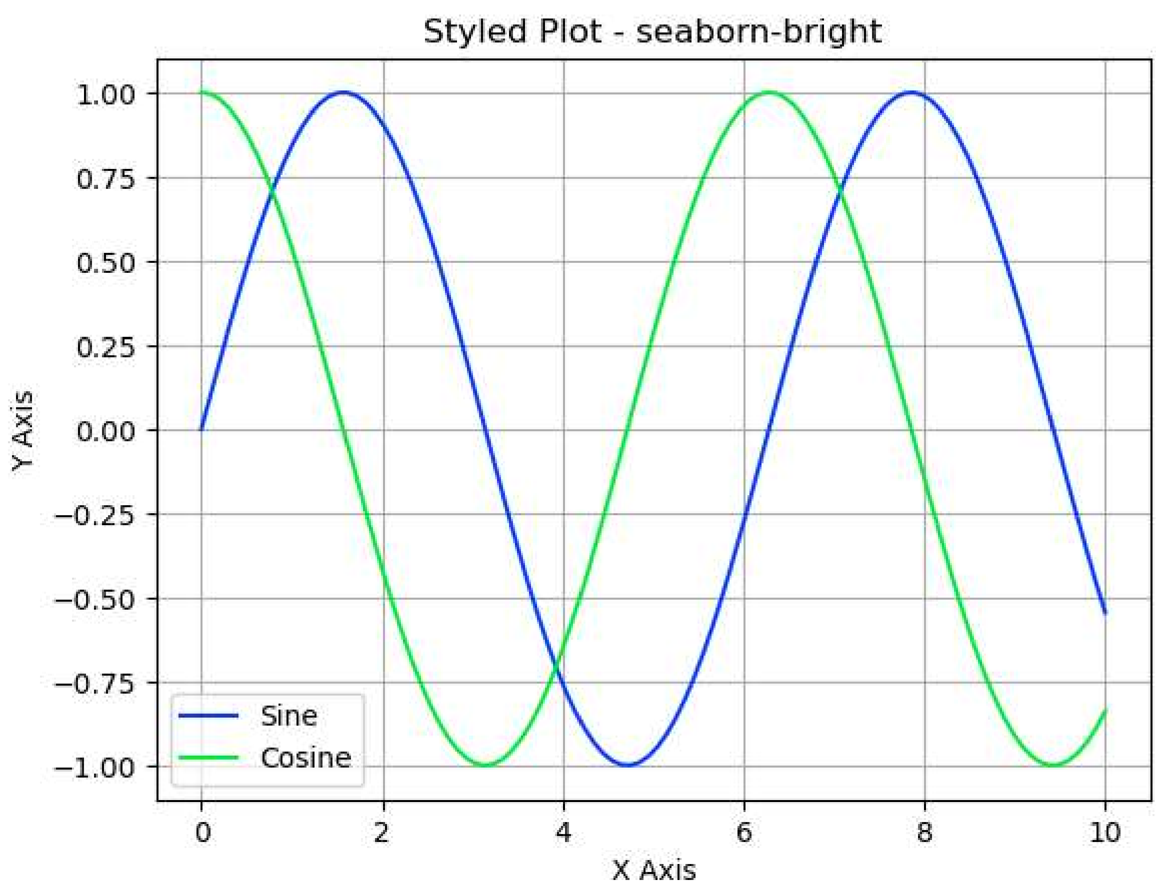
```
plt.style.use('seaborn-v0_8-bright')

# Create data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Plot the data
plt.plot(x, y1, label='Sine')
plt.plot(x, y2, label='Cosine')

plt.title('Styled Plot - seaborn-bright')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.grid(True)
plt.legend()
```

Out[115…

```
<matplotlib.legend.Legend at 0x1e5bc5a74d0>
```

In [117…

```
plt.show()
```

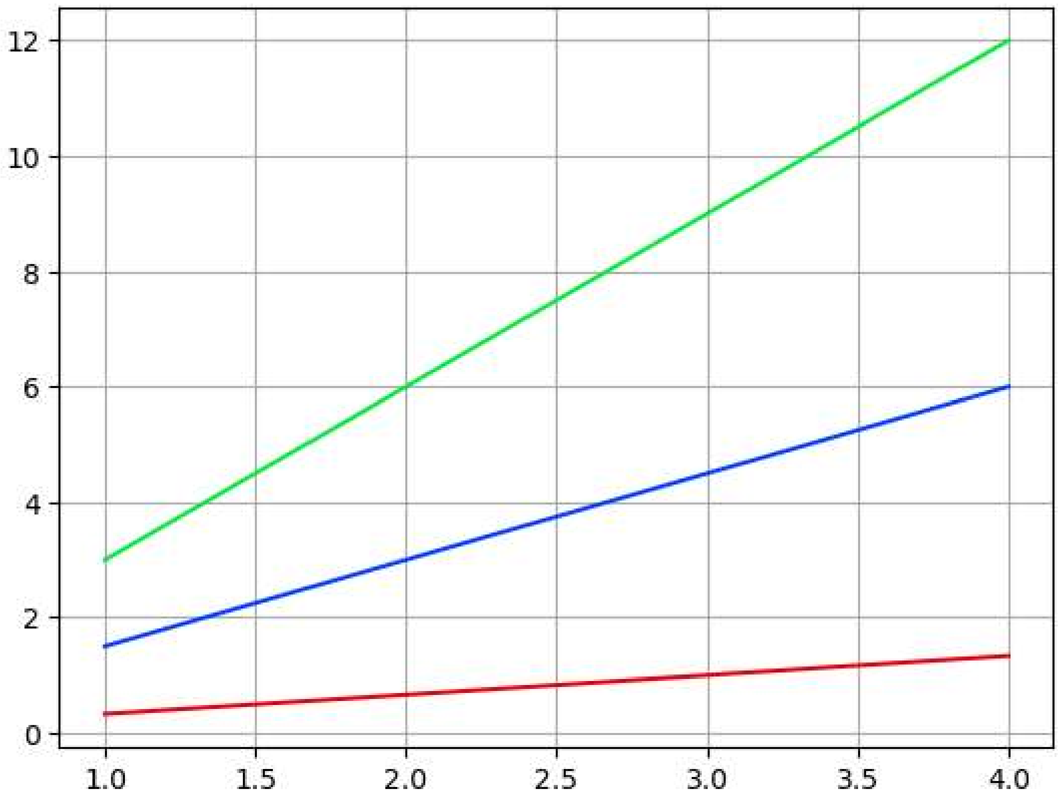


Adding a grid

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.

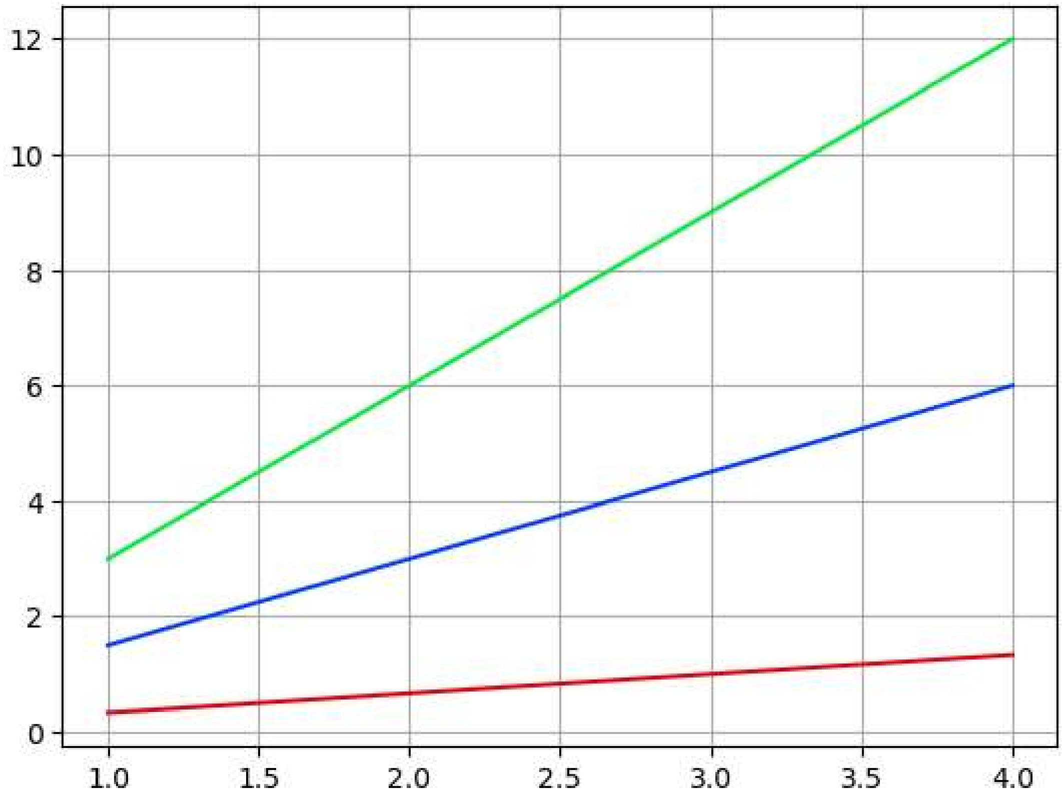
In [128…

```
#P-1

x15 = np.arange(1, 5)

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

plt.grid()#By default it is true

plt.show()
```



In [130…

```
#P-2

x15 = np.arange(1, 5)

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

plt.grid(True)

plt.show()
```

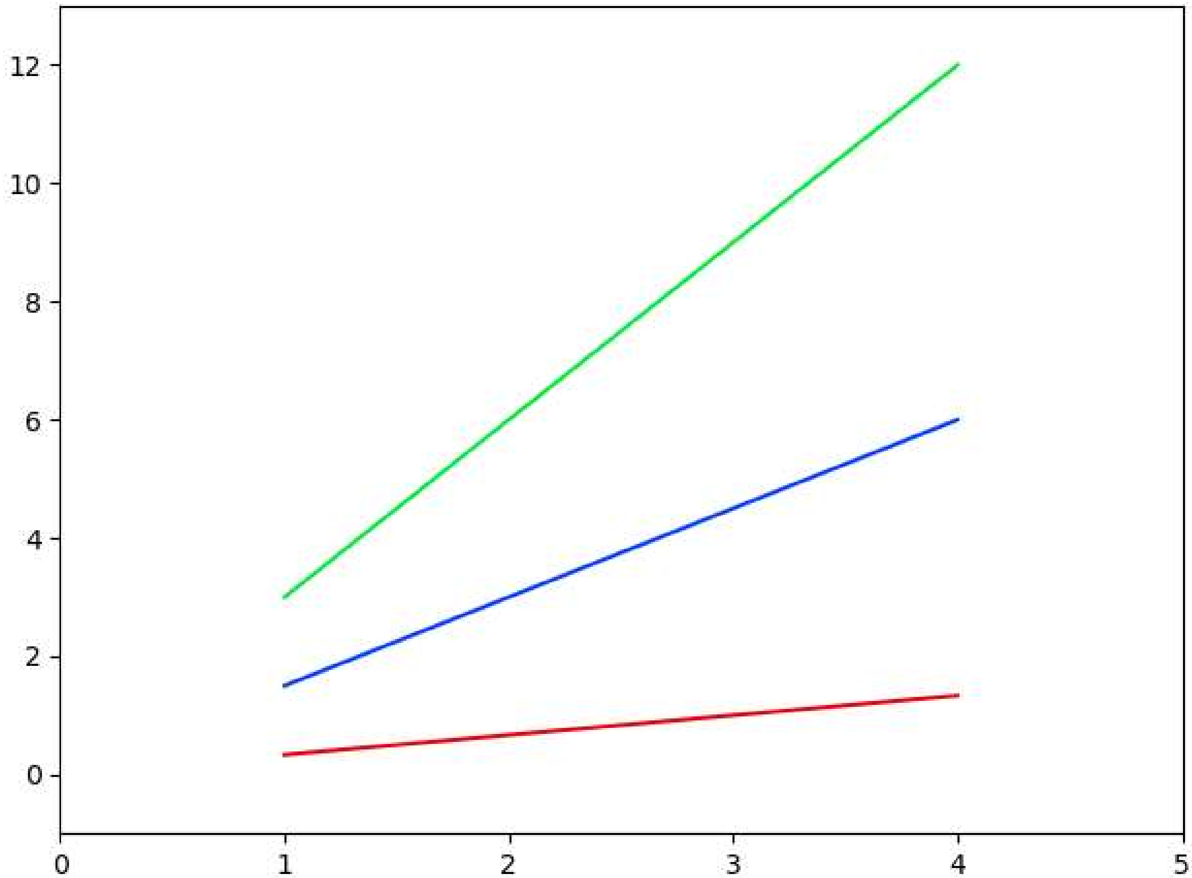


In [ ]:
```
#P-3
x15 = np.arange(1, 5)

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

plt.grid(False)

plt.show()
```

Handling Axes

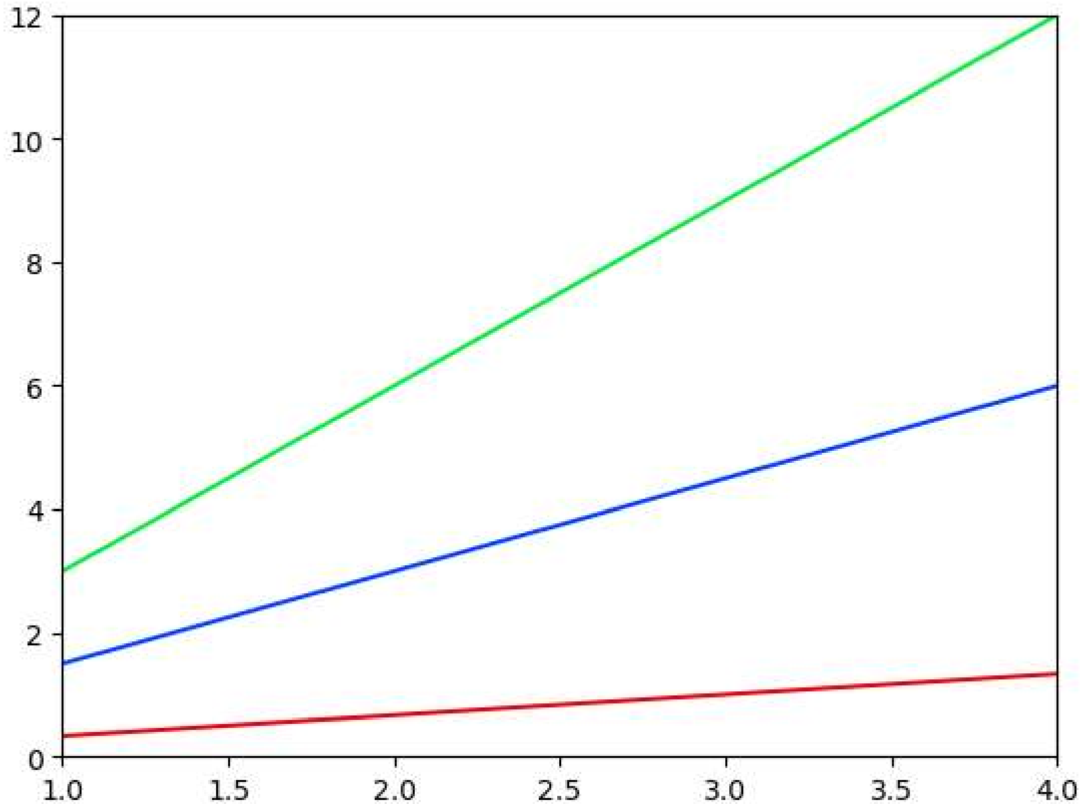
Matplotlib automatically sets the limits of the plot to precisely contain the plotted datasets. Sometimes, we want to set the axes limits ourself. We can set the axes limits with the axis() function as follows:-

In [147...
```
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)#created three lines

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

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

plt.tight_layout()

plt.show()
```



In [157…

```
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])

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:-

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.

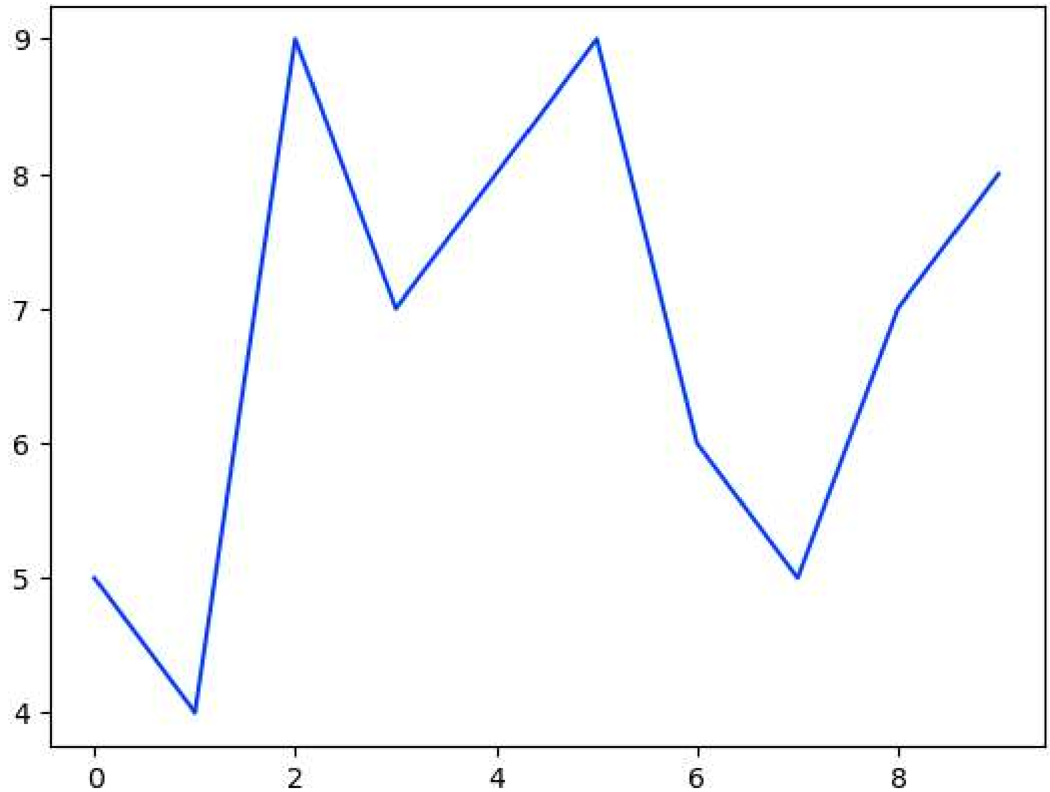
We can pass arguments(in the form of lists) to the ticks functions. The arguments are:-

Locations of the ticks

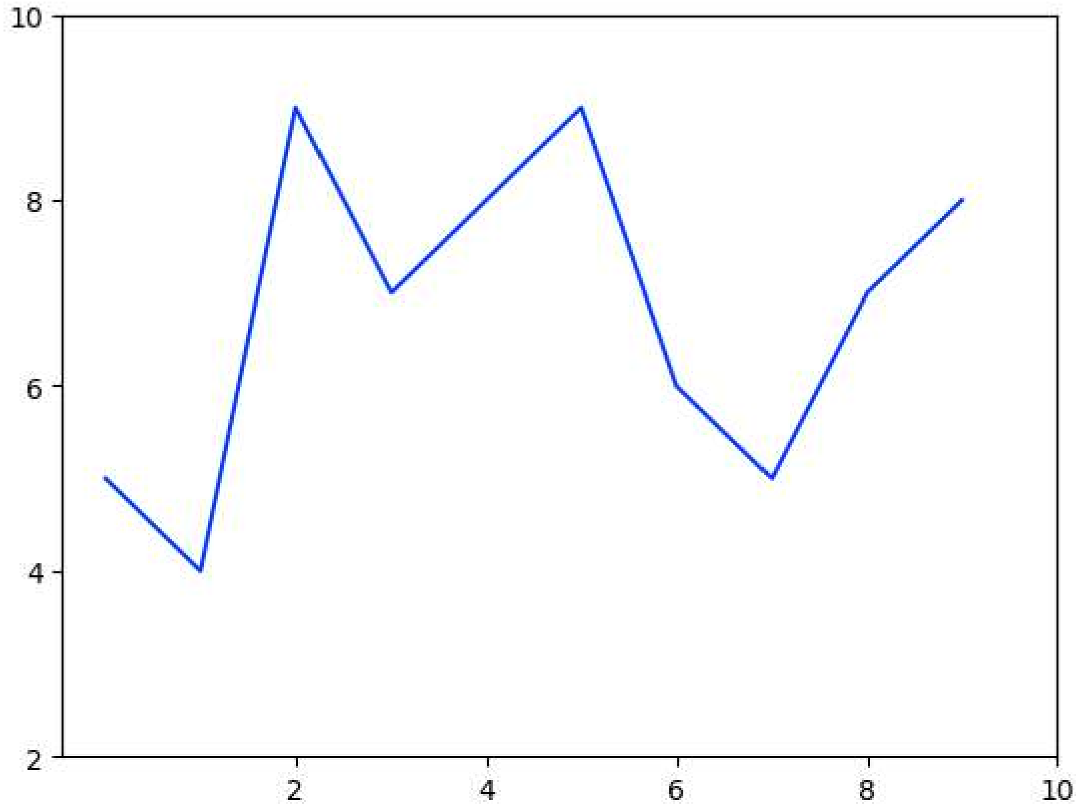
Labels to draw at these locations.

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

In [160…
```
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()
```



In [162…
```
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()
```

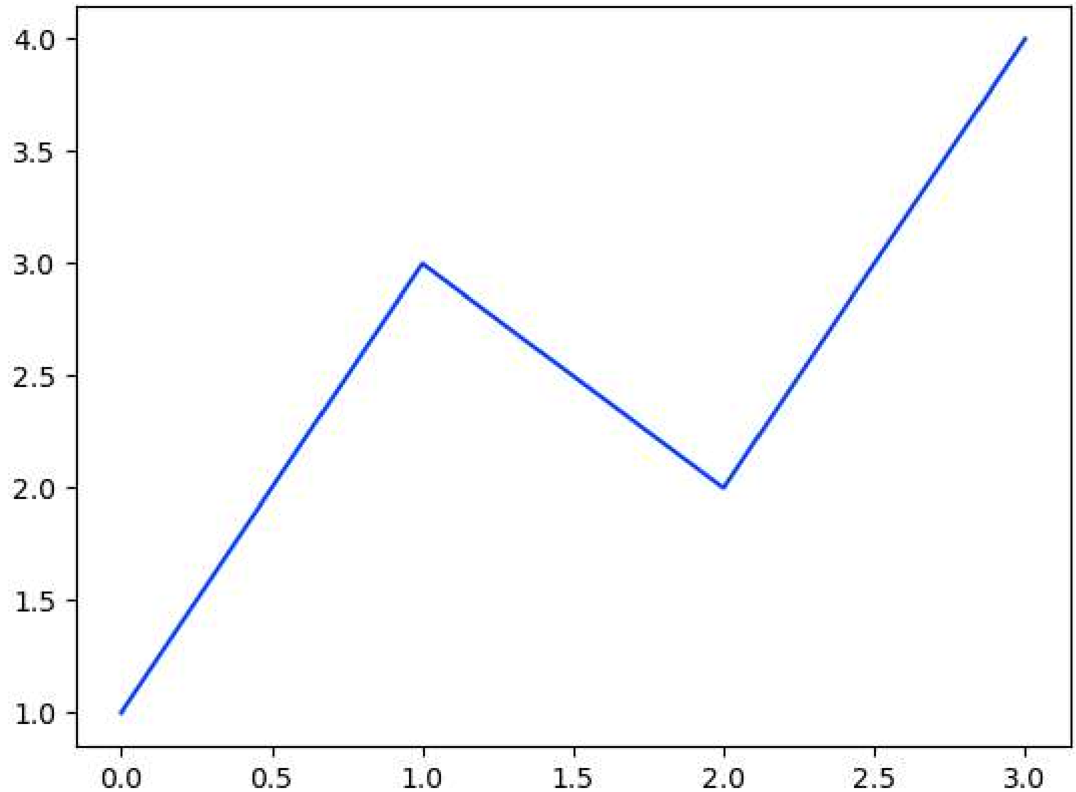


ADDING LABELS

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

In [167…

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

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

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

plt.show()
```

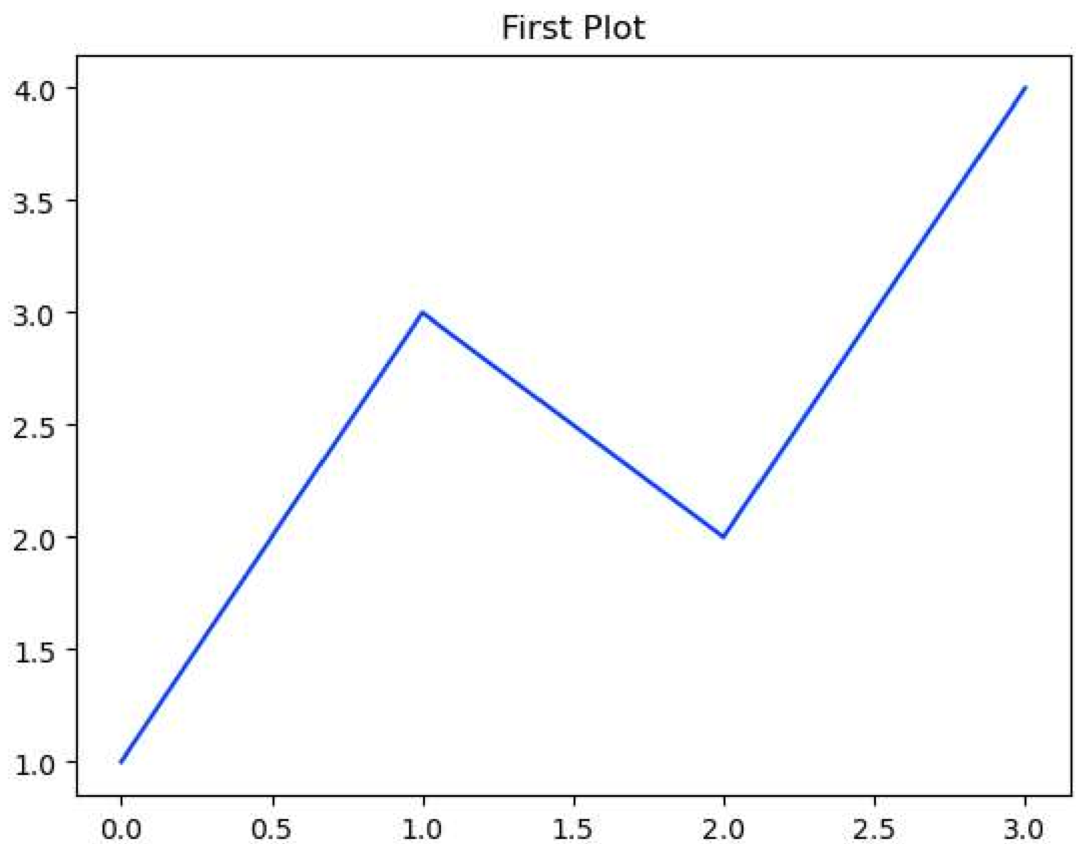


ADDING A TITRLE

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

In [170…

```
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.

ADDING A LEGEND

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

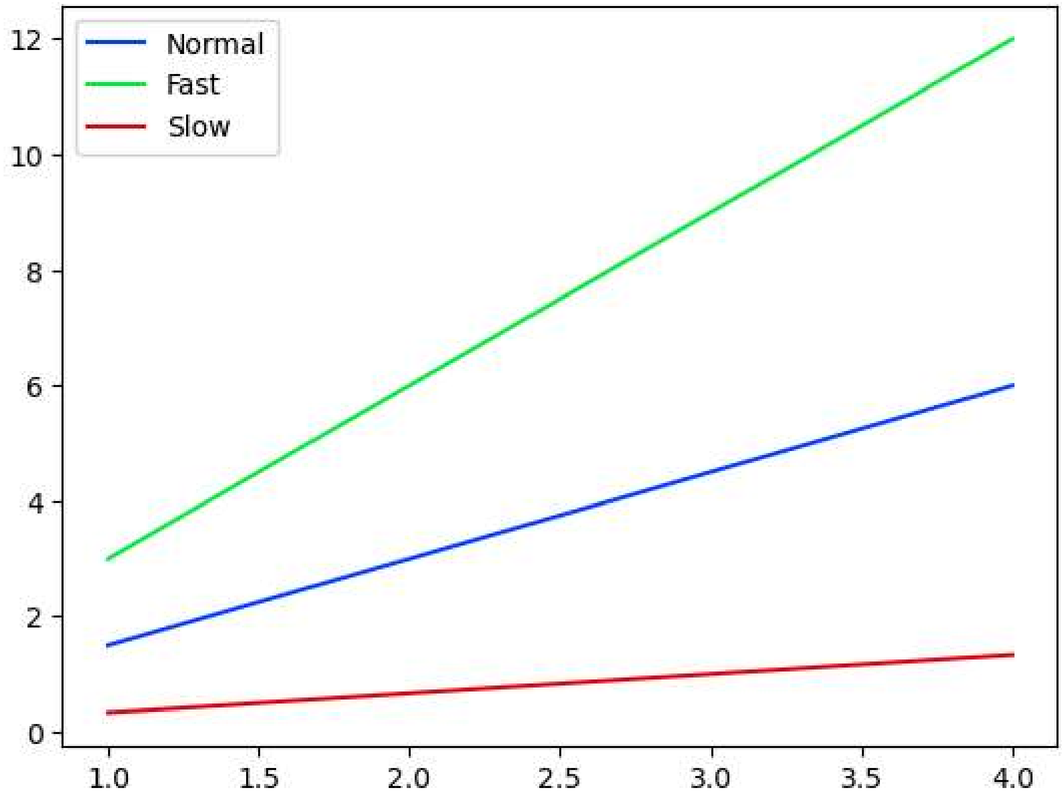
Legends for curves in a figure can be added in two ways. One method is to use the legend method of the axis object and pass a list/tuple of legend texts as follows:-

In [173…

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

fig, ax = plt.subplots()#object-oriented way of plotting in Matplotlib

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

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

In [175…

```
plt.show()
```



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.

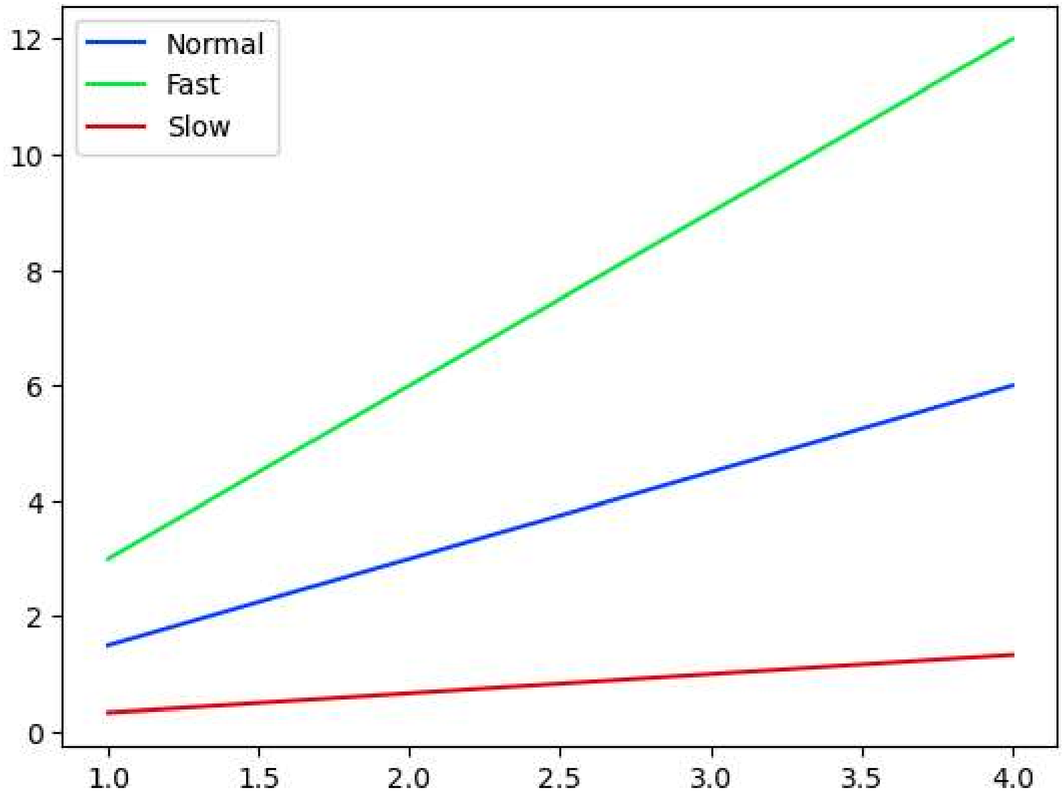
The advantage of this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly. It can be achieved by executing the code below:-

In [177…

```
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();
```

In [179…

```
plt.show()
```



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

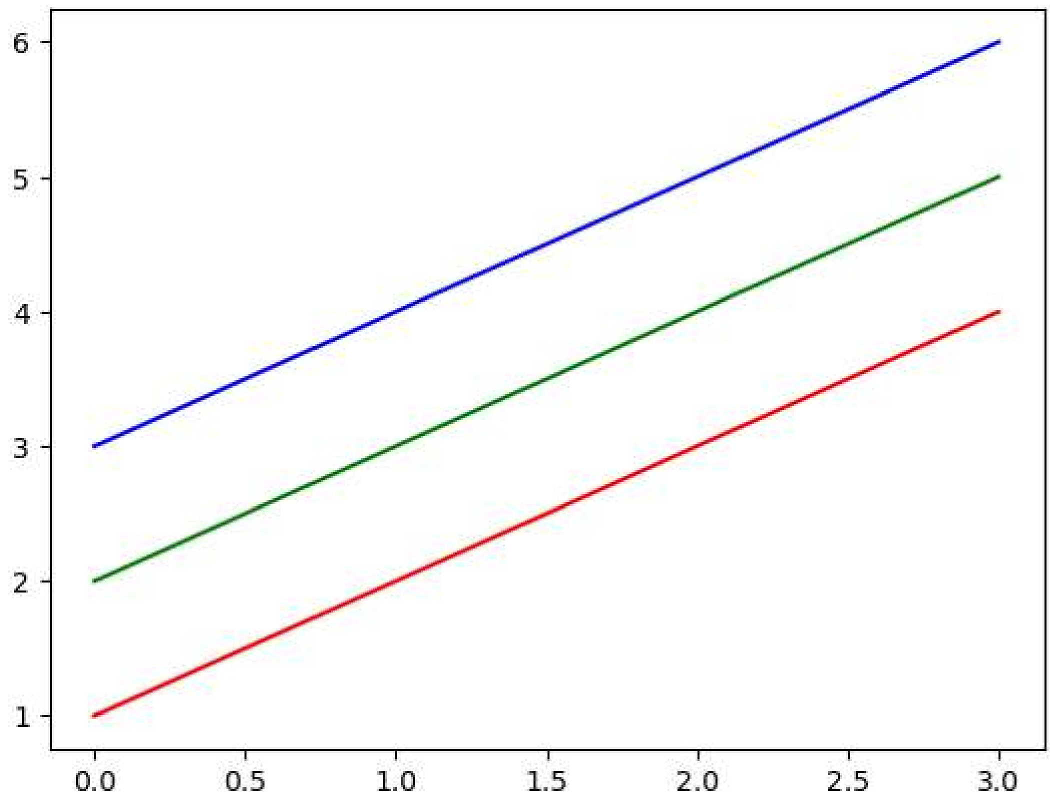
ax.legend(loc=10) # center

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.

In [184…

```
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 are 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)

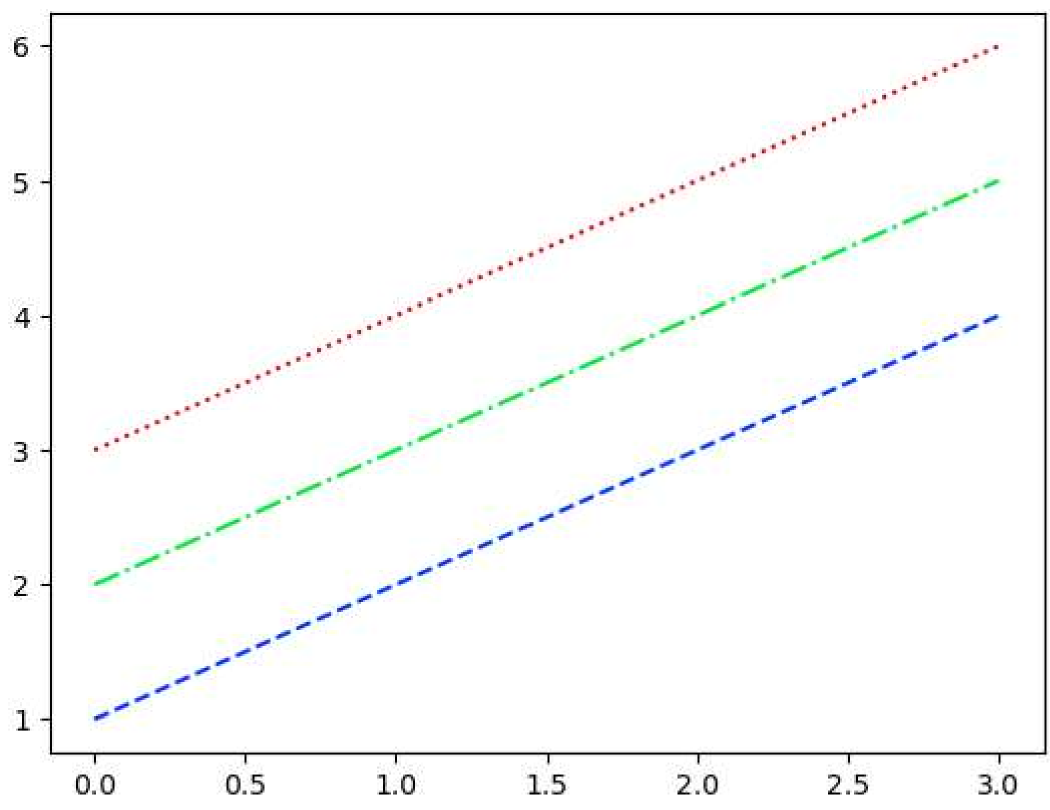
• Grayscale intensity, in string format such as ‘0.7’.

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.

In [187…

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

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

plt.show()
```



SUMMARY

In this project, I discuss Matplotlib (the basic plotting library in Python) and throw some light on various charts and customization techniques associated with it.

In particular, I discuss Matplotlib object hierarchy, Matplotlib architecture, Pyplot and Object-Oriented architecture. I also discuss subplots which is very important tool to create graphics in Matplotlib.

Then, I discuss various types of plots like line plot, scatter plot, histogram, bar chart, pie chart, box plot, area chart and contour plot.

Finally, I discuss various customization techniques. I discuss how to customize the graphics with styles. I discuss how to add a grid and how to handle axes and ticks. I discuss how to add labels, title and legend. I discuss how to customize the charts with colours and line styles.