Matplotlib-Final

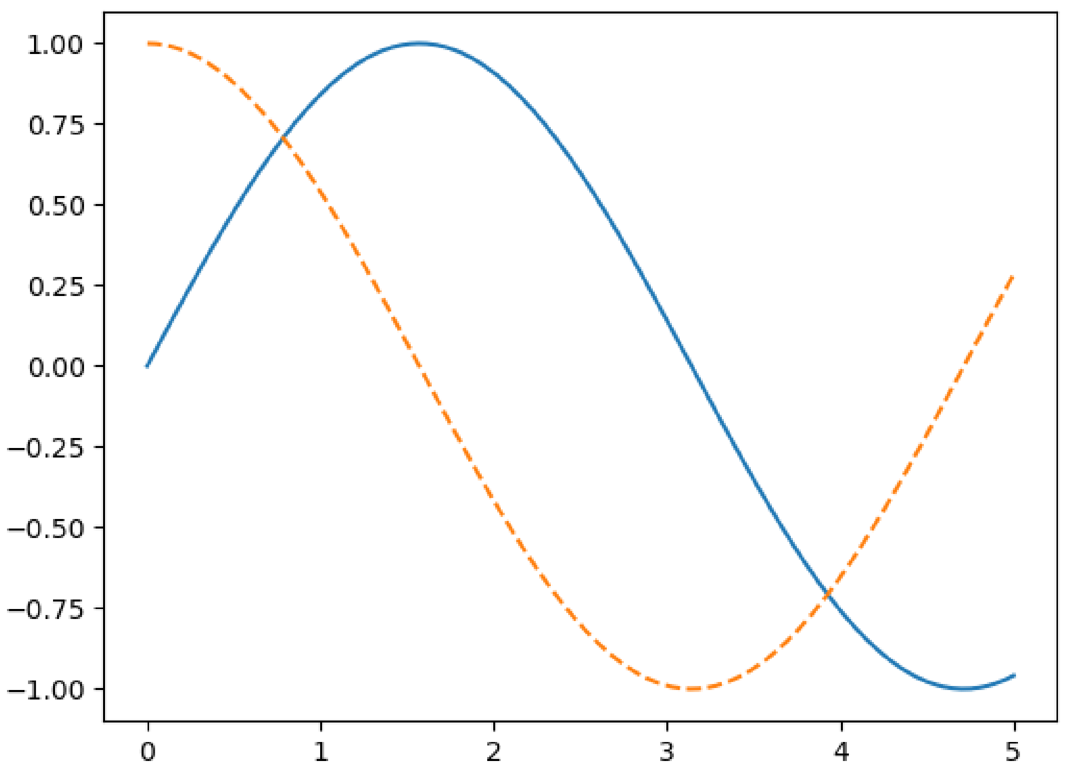
February 18, 2025

1 Matplotlib

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

[ ]:
```
# Displaying Plots in MatploLib

x1 = np.linspace(0.0, 5.0, 100)

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

plt.plot(x1,np.sin(x1),'-') #plot 1
plt.plot(x1,np.cos(x1),'--') #plot 2
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x1dfdf123190>]
```



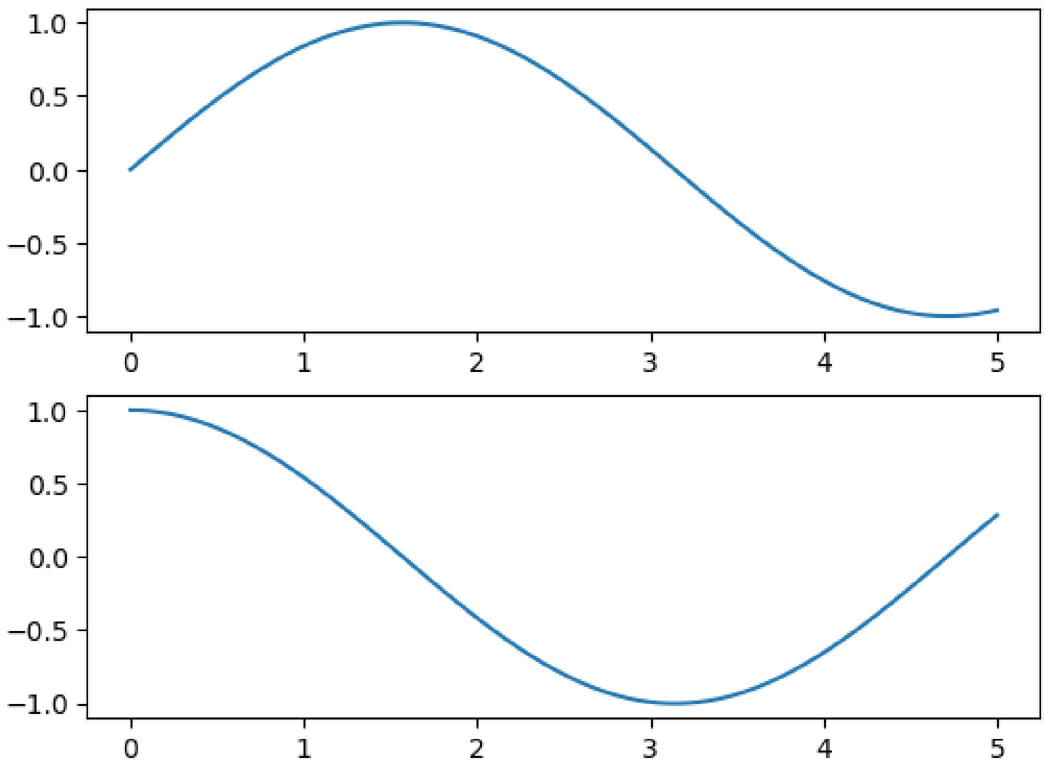
1.1 Matplotlib API Overview

Matplotlib has two API 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.

the following code produces sin and cosine curves using Pyplot API

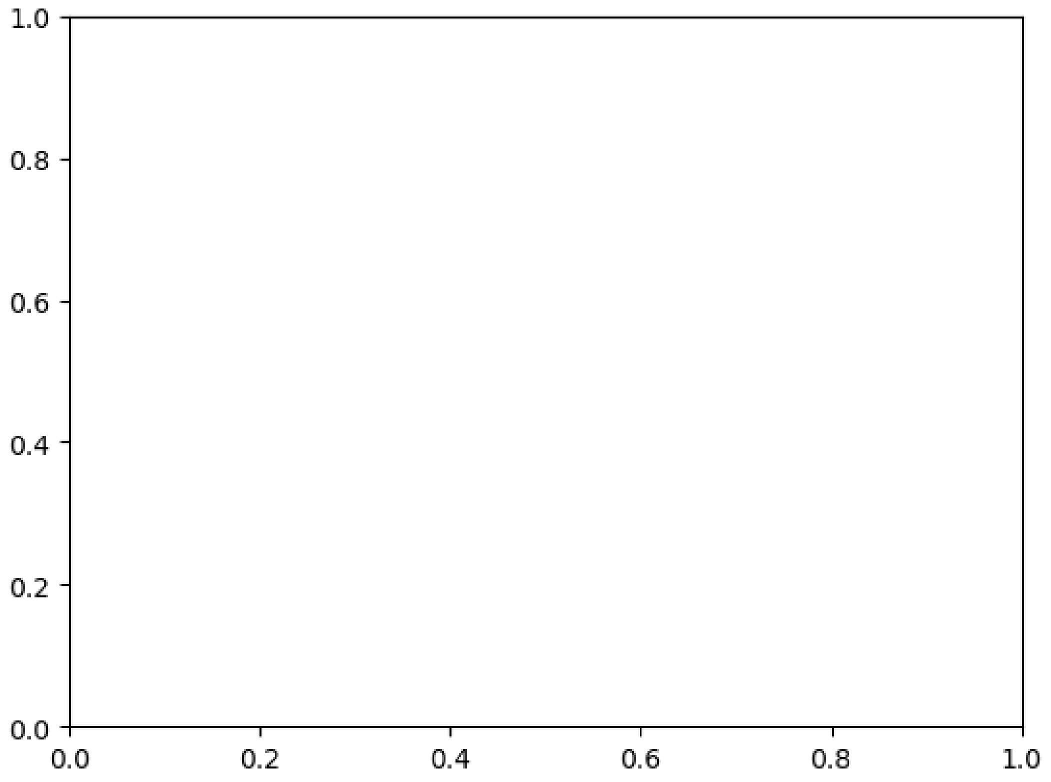
[4]:
```
# Create a plot Figure
plt.figure()

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

# Create the second panel and set current axis
plt.subplot(2,1,2) # (rows, columns, Axes number)
plt.plot(x1,np.cos(x1))
```

[4]:
```
[<matplotlib.lines.Line2D at 0x1dfe0f2d650>]
```



[5]:
```
# Get current figure and axes using plt.gcf() and plt.gca()

plt.gcf() # get current figure details

plt.gca() # get current axes details
```

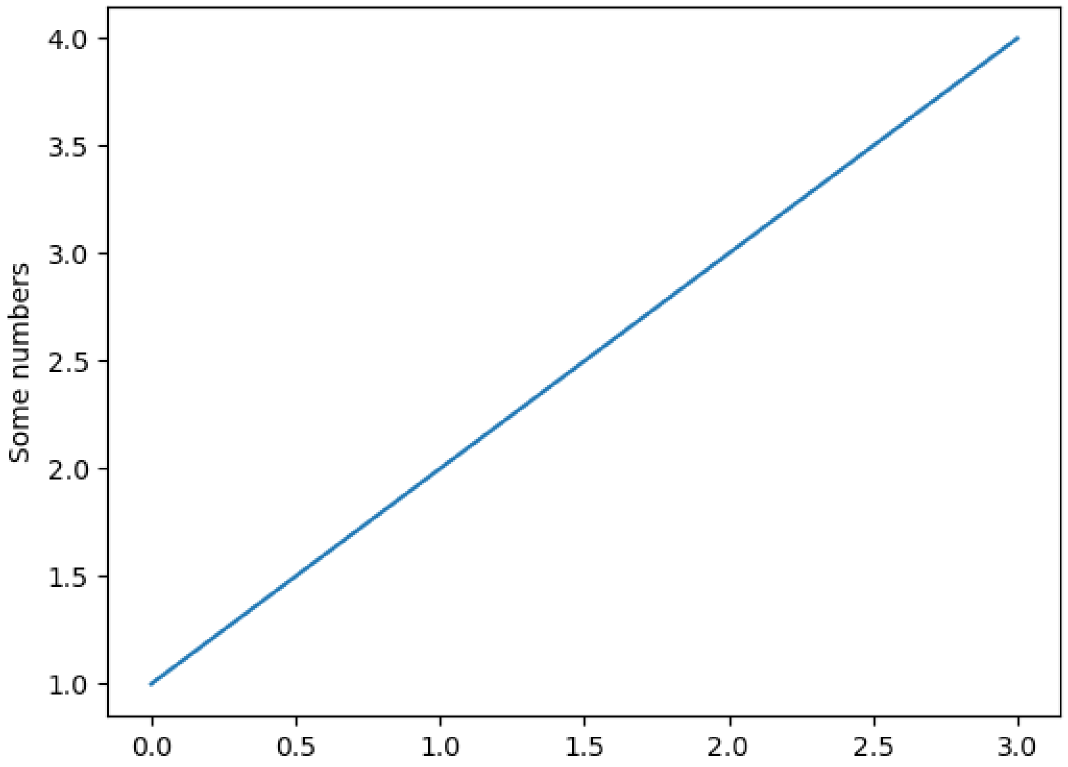
[5]:
```
<Axes: >
```



1.2 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].

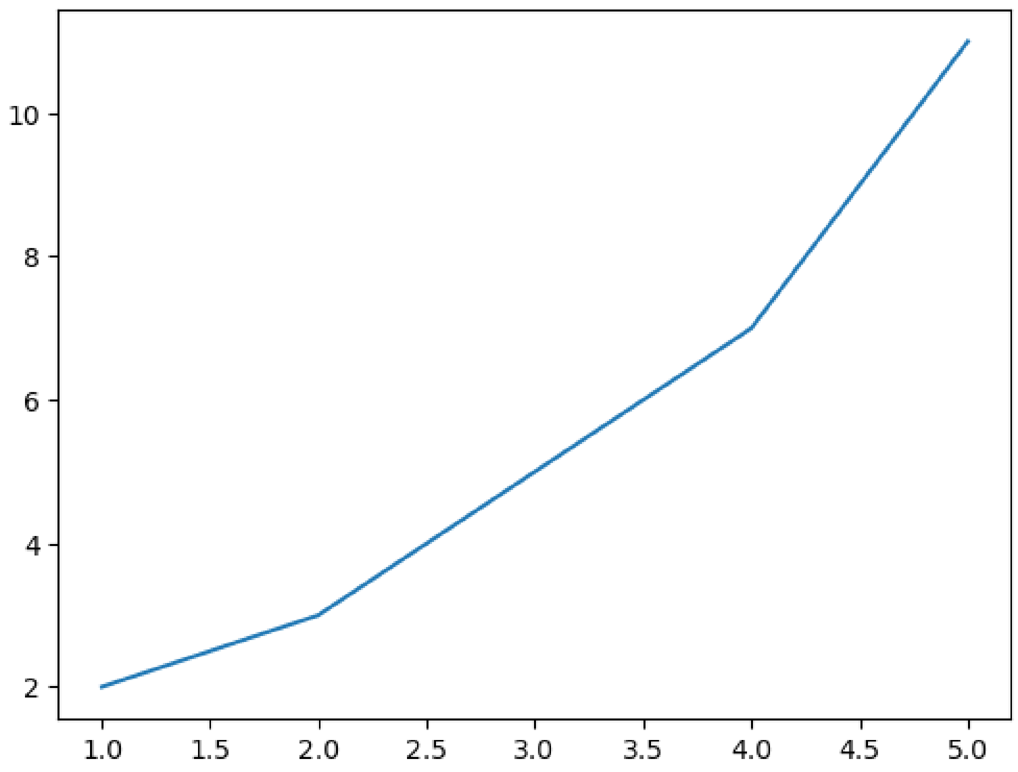
[6]:
```
plt.plot([1,2,3,4])
plt.ylabel('Some numbers')
plt.show()
```



1.2.1 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:-

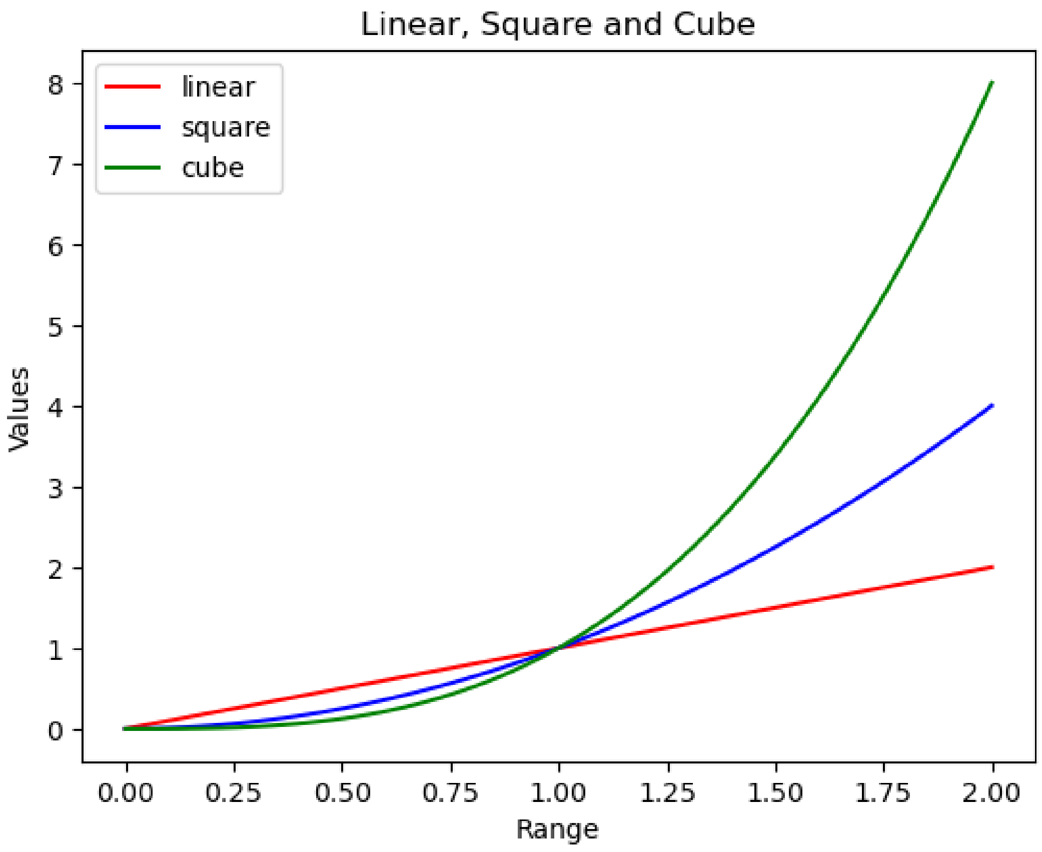
[7]:
```
plt.plot([1,2,3,4,5],[2,3,5,7,11])
plt.show()
```



1.2.2 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:

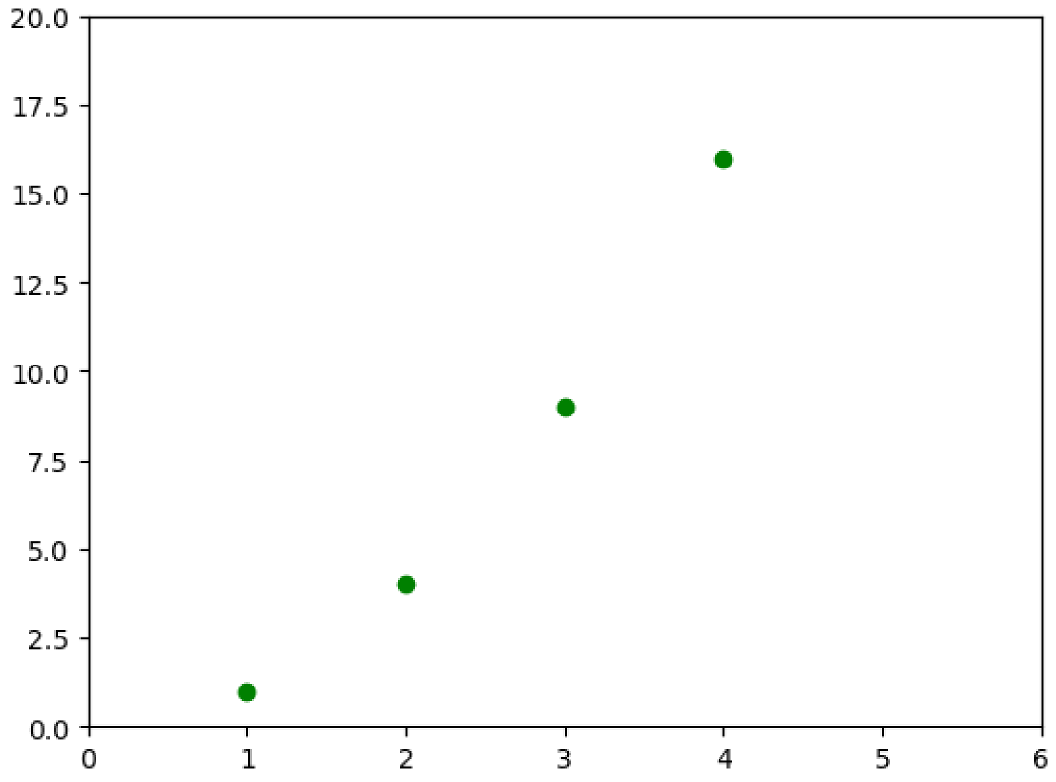
[8]:
```
x = np.linspace(0,2,100)
plt.plot(x,x,label = 'linear',c='r') # c is color
plt.plot(x,x**2,label= 'square',c='b') # label is used to give name to the plot
plt.plot(x,x**3,label= 'cube',c='g') # x is the X-axis and x**3 is the Y-axis
plt.xlabel('Range') # X-axis label name
plt.ylabel('Values') # Y-axis label name
plt.title('Linear, Square and Cube') # Title of the plot
plt.legend() # Show the legend
plt.show() # Show the plot
```



1.2.3 Formatting the style of your 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:-

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

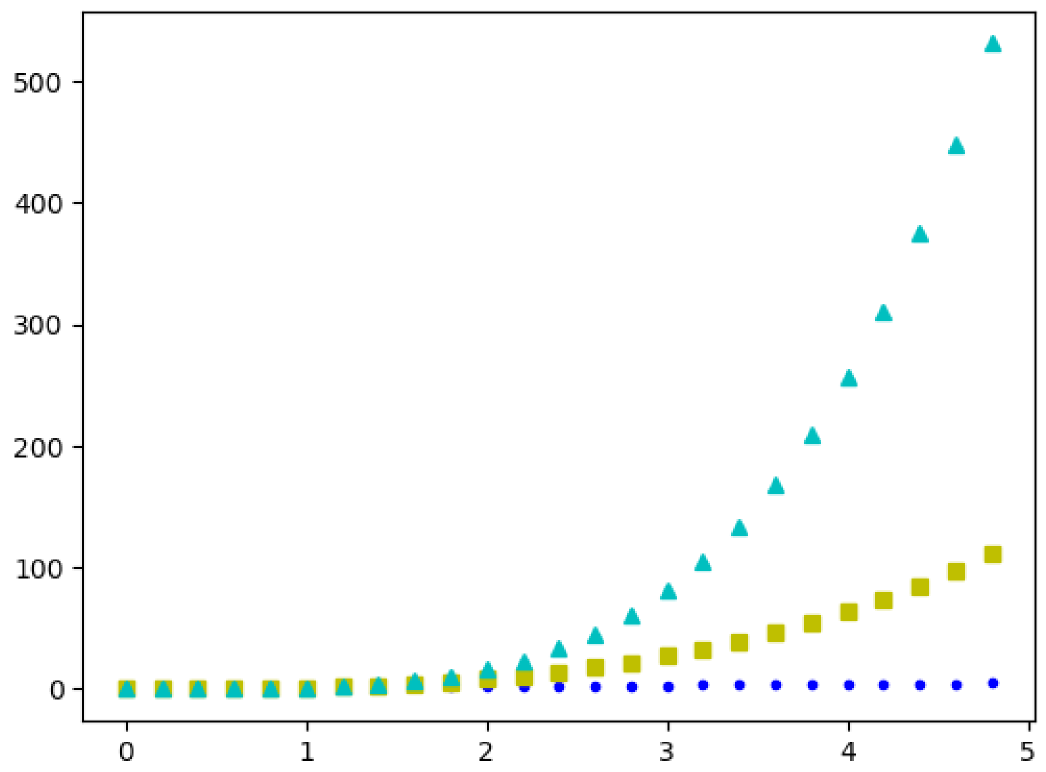


The **axis()** command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

1.2.4 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.

[10]:
```
# evenly sampled time at 200ms intervals
t = np.arange(0.,5.,0.2)
# blue dashes, yellow squares and cyan triangles
plt.plot(t,t,'b.',t,t**3,'ys',t,t**4,'c^') # we can plot multiple plots in a␣
 ↪single plot
plt.show()
```



1.3 Object-Oriented Interface

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.

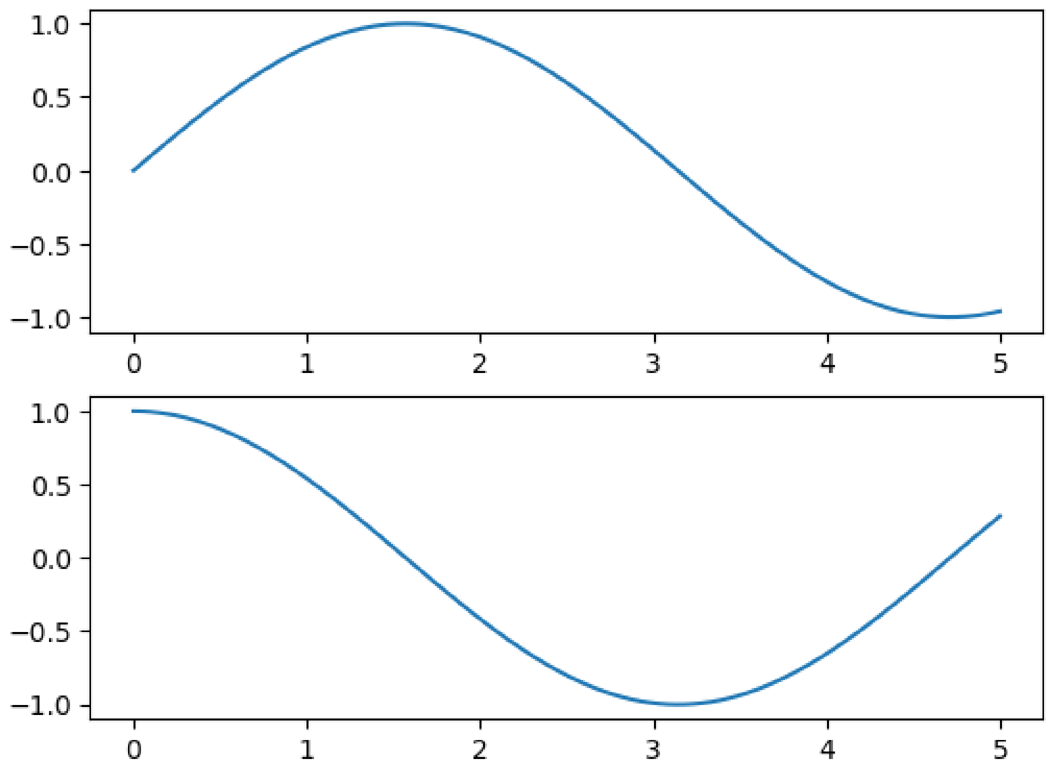
Figure is the top level container for all the plot elements. We can think of the **Figure** object as a box-like container containing one or more **Axes**.

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.

```
[11]: # Create a figure and an axis using plt.subplots()
      fig,ax = plt.subplots(2) #this will create 2 plots with 2 rows and 1 column

       # Call plot() method on the appropriate object
      ax[0].plot(x1,np.sin(x1)) # plot 1 at axis 0
      ax[1].plot(x1,np.cos(x1)) # plot 2 at axis 1
```

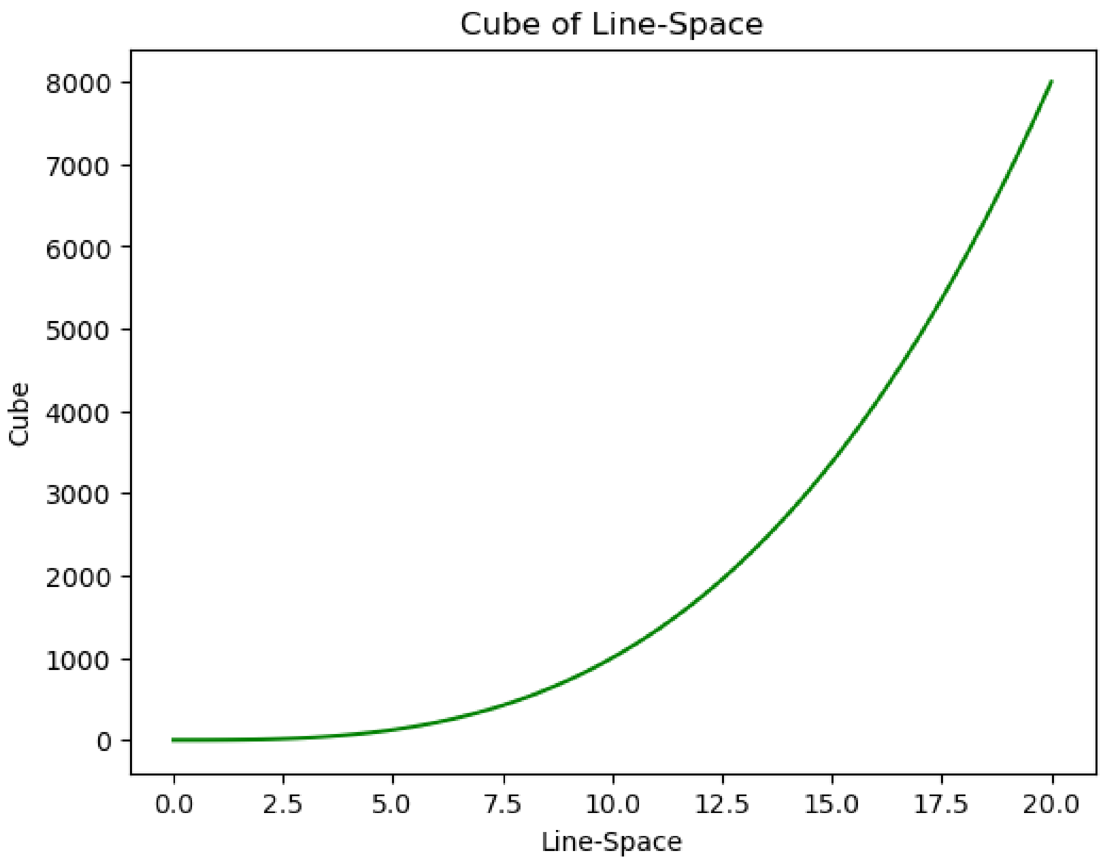
```
[11]: [<matplotlib.lines.Line2D at 0x1dfe1955350>]
```



1.3.1 Object and Reference

- In Python, variables are references to objects in memory.
- When we set a variable equal to another variable,both variables
  point to the object in memory.
- If we change one variable, the other variable will reflect this change.
- If we want to create a new object in memory, we must use the copy() method.

[12]:
```
# Create a container for the plot using plt.figure()
fig = plt.figure()
x2 = np.linspace(0,20,100)
y2 = x2**3
axes = fig.add_axes([0.5,0.5,0.8,0.8]) # space from left, space from bottom,␣
 ↪width, height)
axes.plot(x2,y2,'g')
axes.set_xlabel('Line-Space')
axes.set_ylabel('Cube')
axes.set_title('Cube of Line-Space')
plt.show()
```



1.3.2 Figure and Axes in Matplotlib

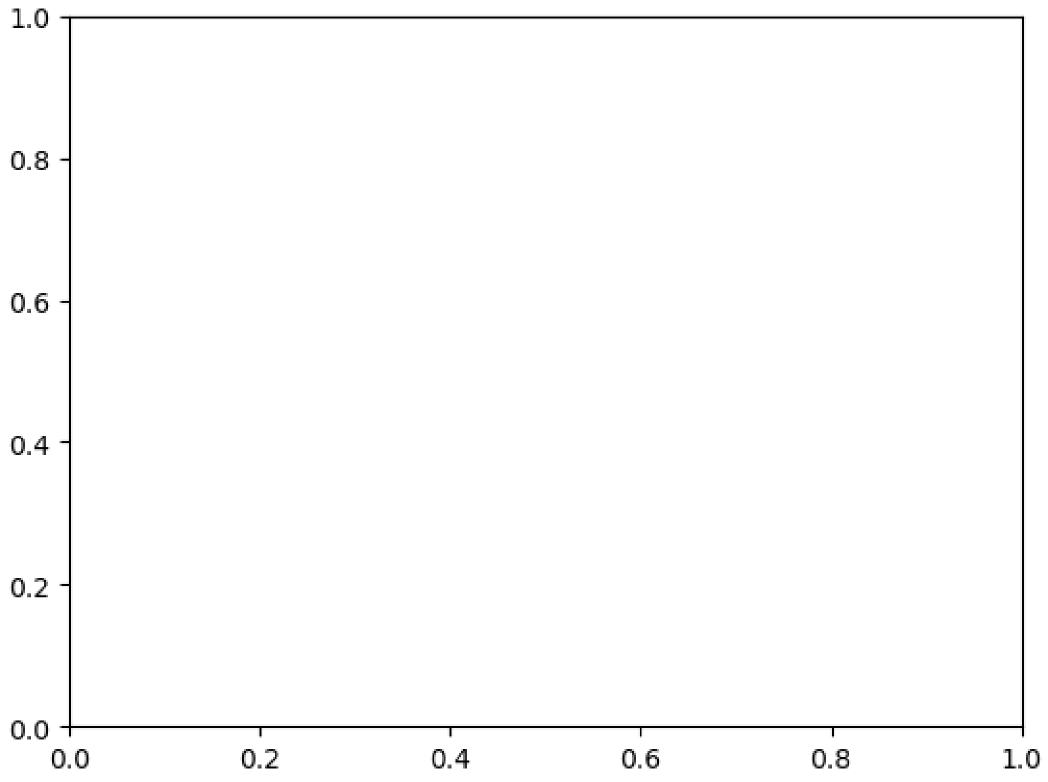
I start by creating a figure and an axes. A figure and axes can be created as follows:

`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.

[13]:
```
# Create a figure and we can customize the size of the figure by passing the␣
 ↪list of values
fig = plt.figure()

# Create an axis and we can customize the size of the axis by passing the list␣
 ↪of values
axes1 = plt.axes()
```



1.4 Figure and Subplots

lots 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:-

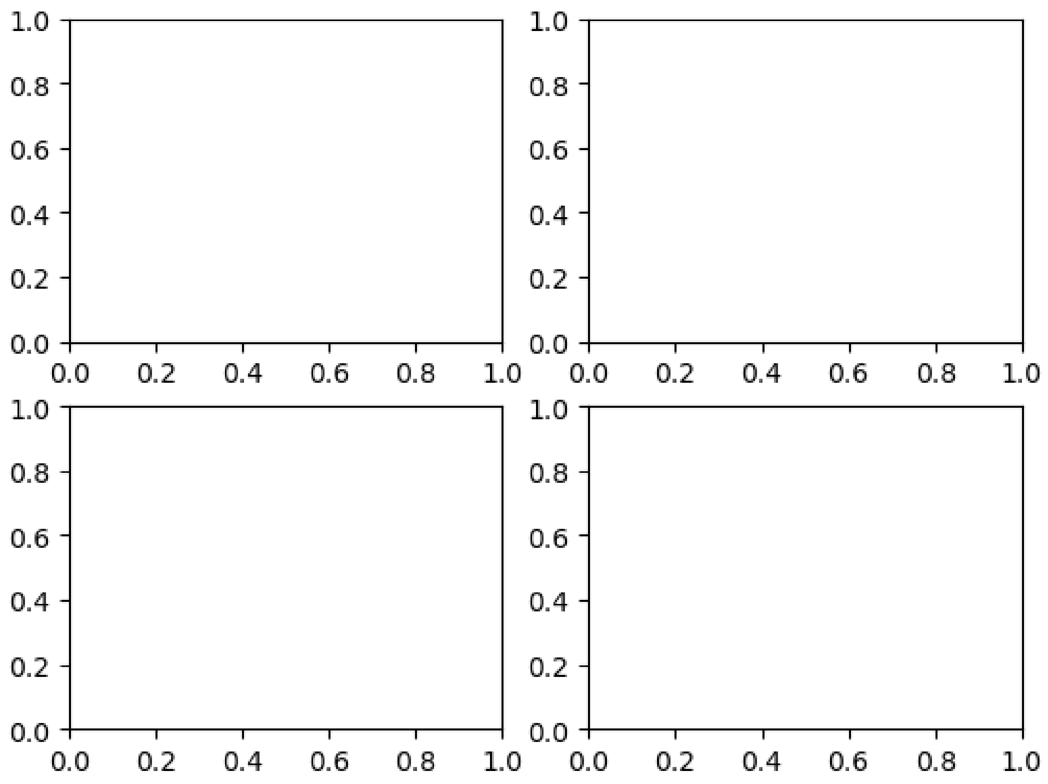
`ax2 = fig.add_subplot(2, 2, 2)`

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

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

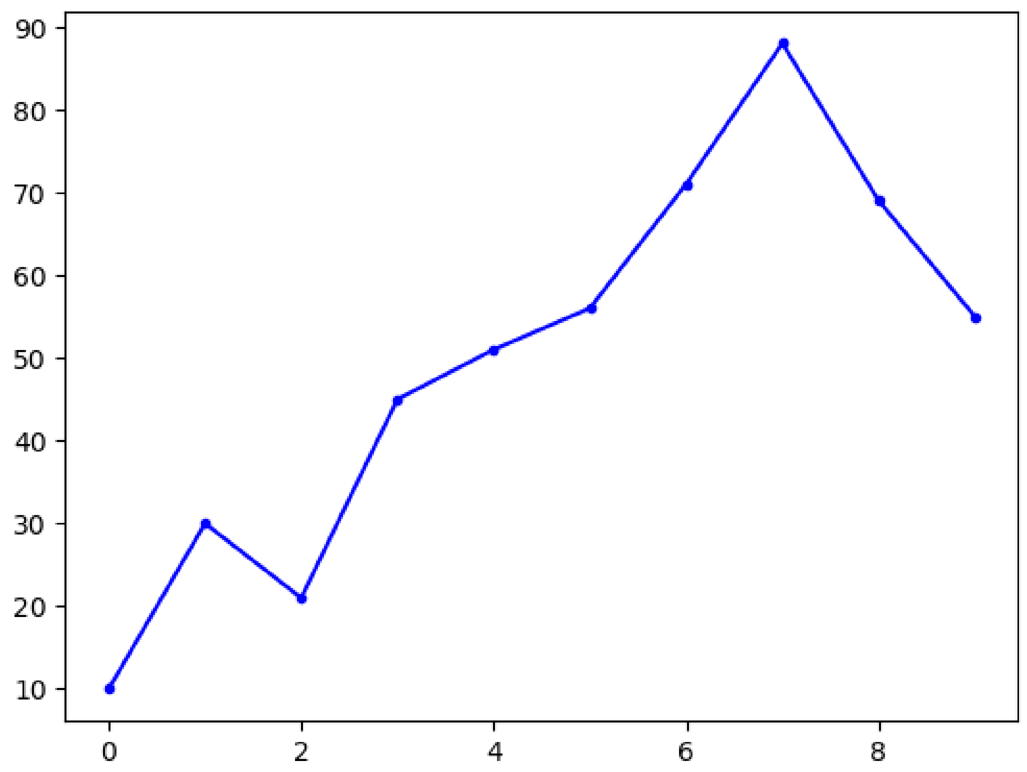
[14]:
```
fig = plt.figure()
axes1 = fig.add_subplot(2,2,1) # 2 rows, 2 columns and 1st plot
axes2 = fig.add_subplot(2,2,2) # 2 rows, 2 columns and 2nd plot
axes3 = fig.add_subplot(2,2,3) # 2 rows, 2 columns and 3rd plot
```

```
axes4 = fig.add_subplot(2,2,4) # 2 rows, 2 columns and 4th plot
```



1.5 First plot with Matplotlib

[15]:
```
plt.plot([10,30,21,45,51,56,71,88,69,55,],'b.-')
plt.show()
```

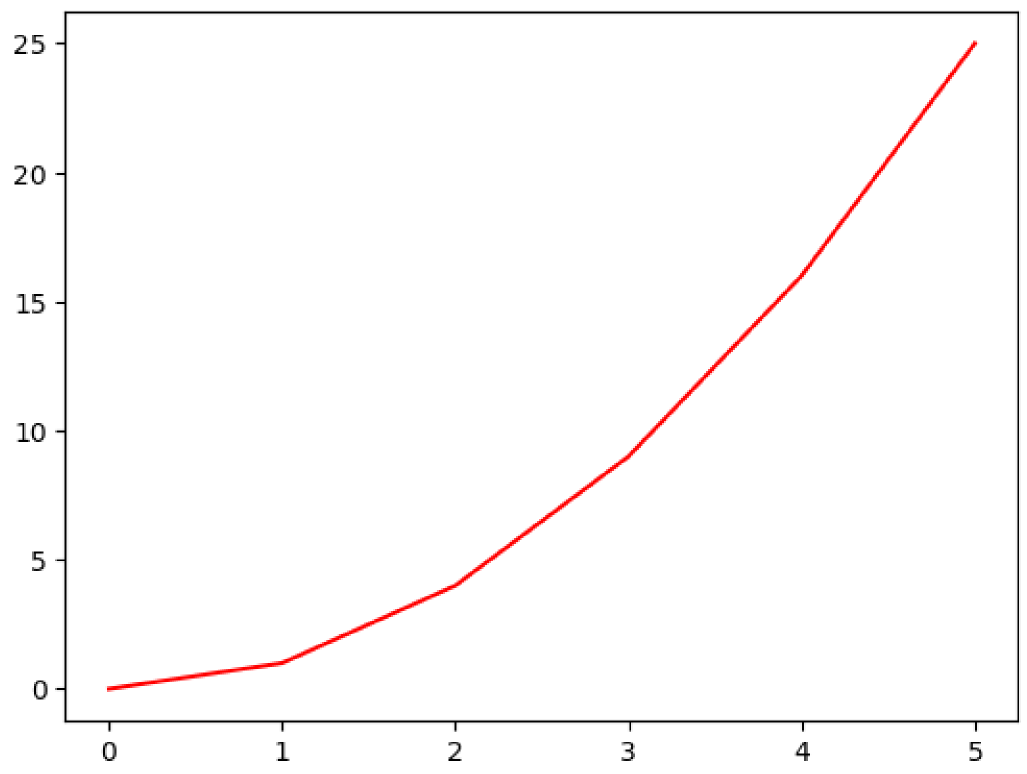


plt.plot([10,30,21,45,51,56,71,88,69,55,],'b.-')

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).

1.5.1 Specify both Lists

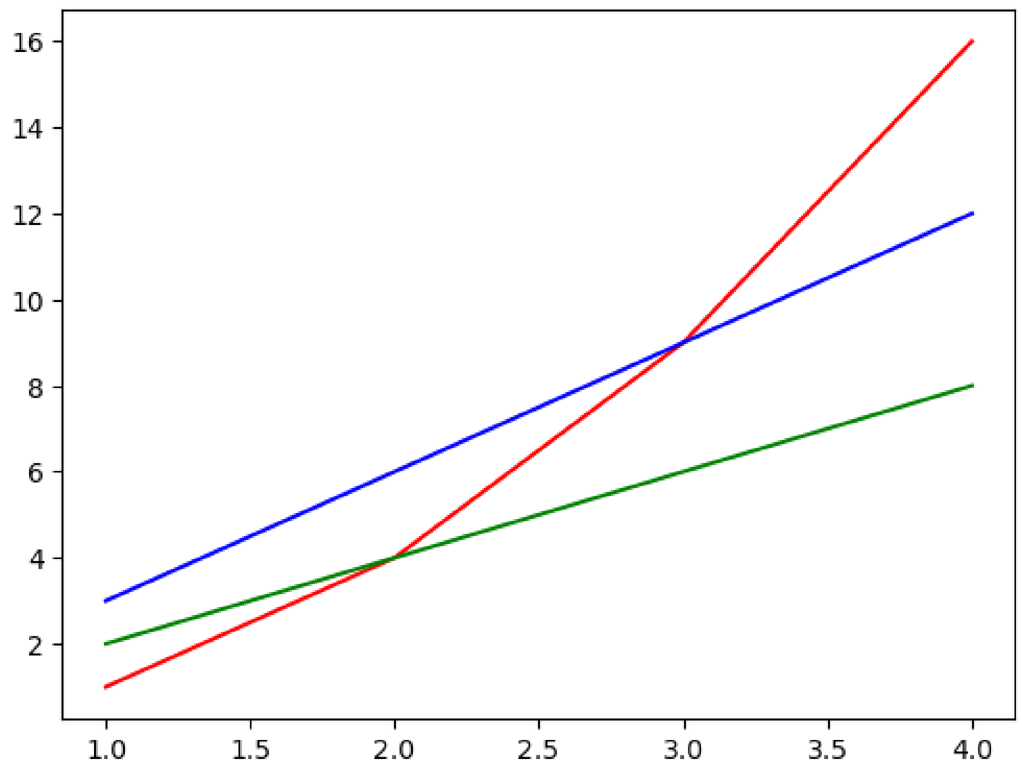
[16]:
```
x3 = range(6)
# we can pass lists of values to the plot() method
plt.plot(x3,[xi**2 for xi in x3],'r-')

plt.show()
```



1.6 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:-

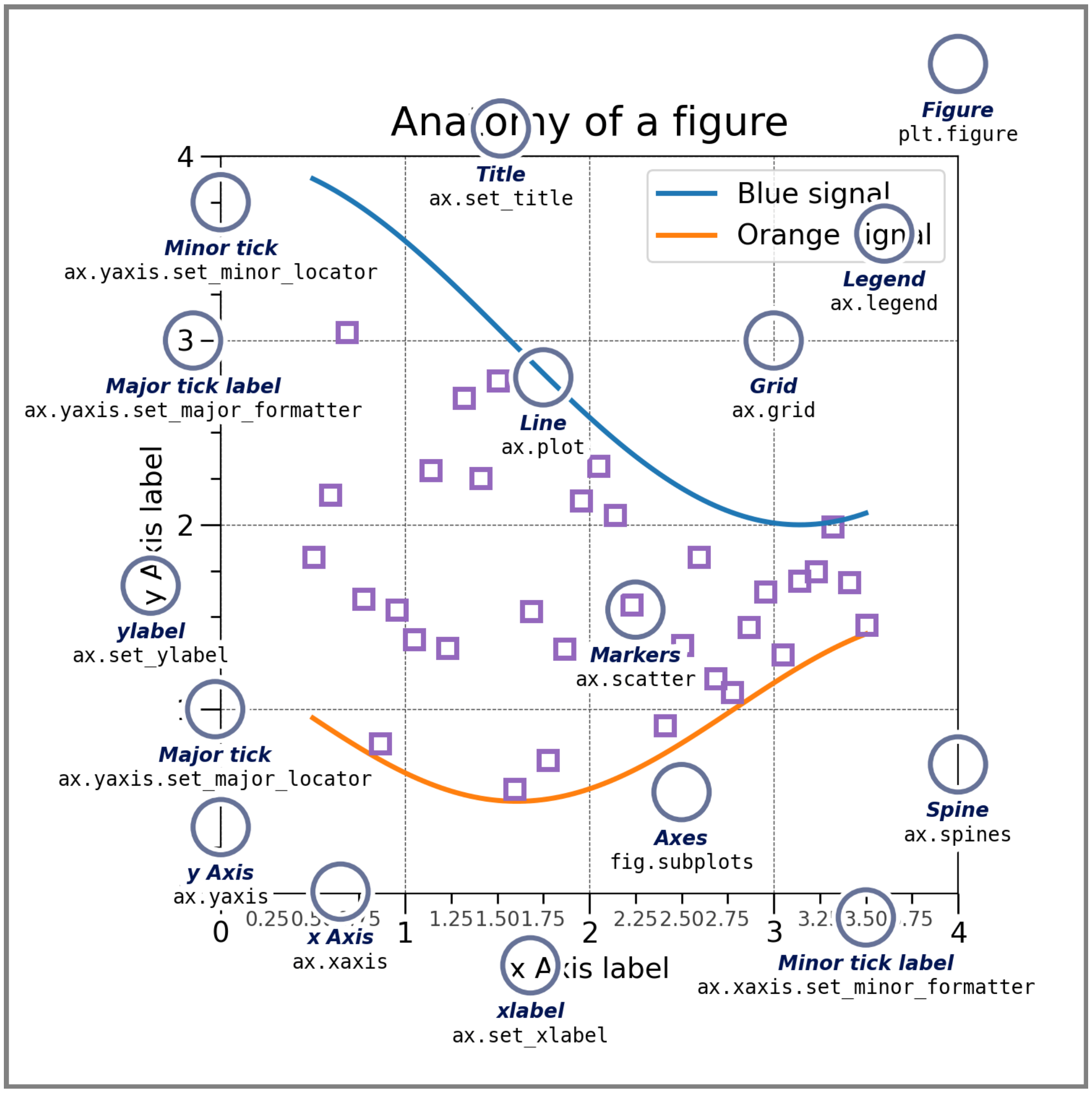
[17]:
```
x = range(1,5)
plt.plot(x,[xy**2 for xy in x],'r-')
plt.plot(x,[xy*3 for xy in x],'b-')
plt.plot(x,[xy*4/2 for xy in x],'g-')
plt.show()
```

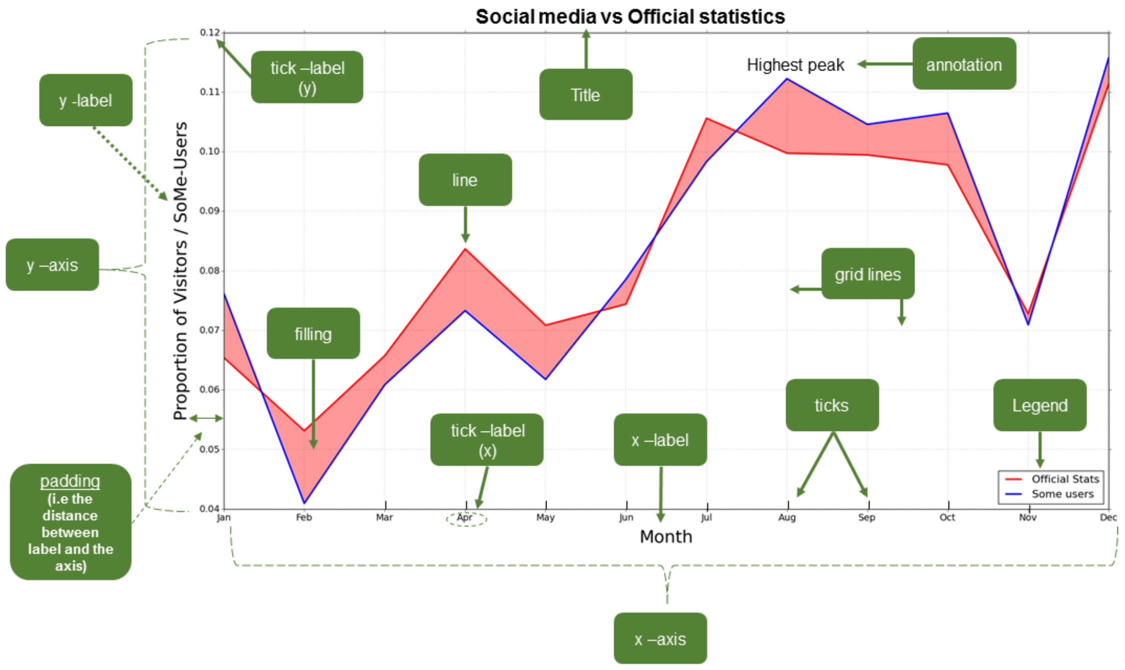


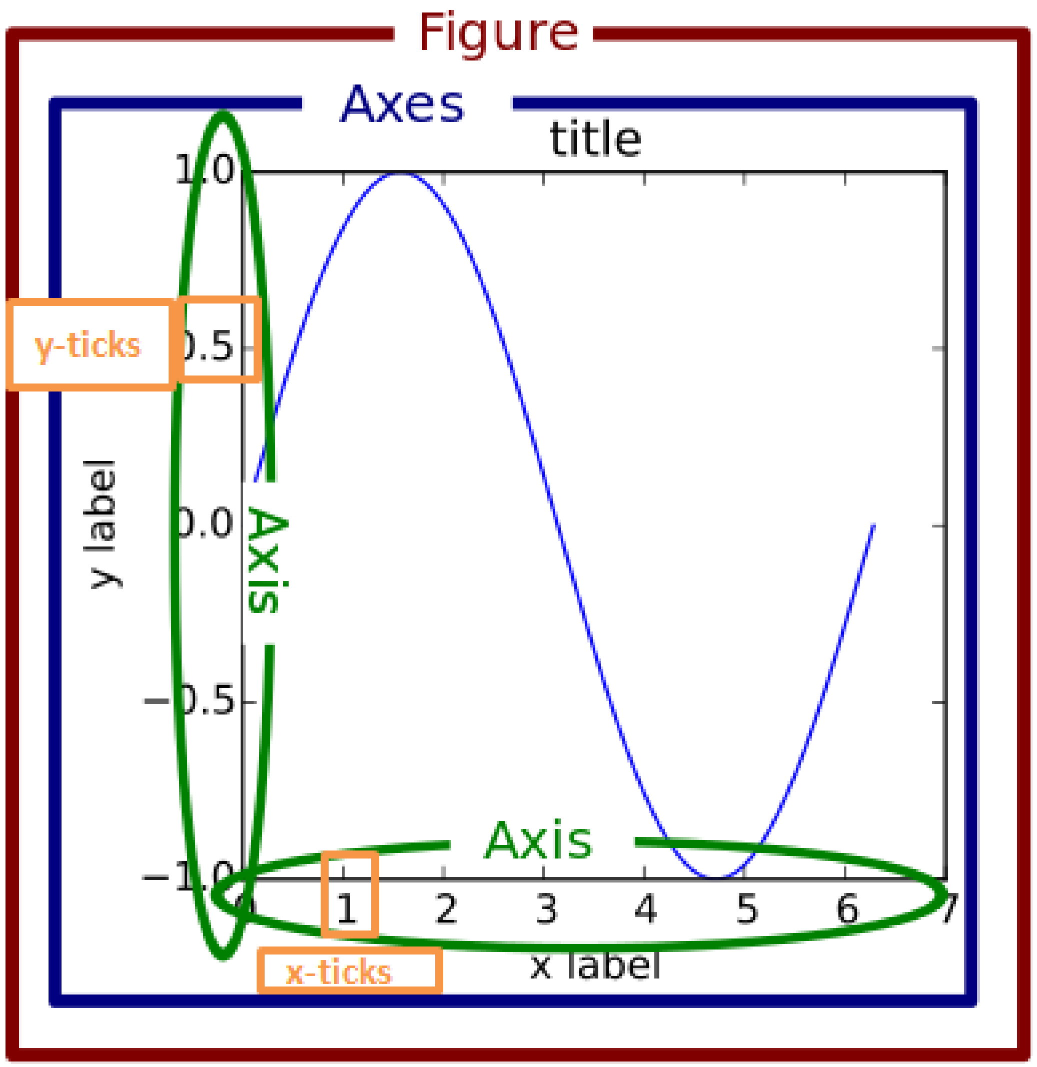
1.6.1 Parts of a Plot

[18]:
```python
# To import image in python
from IPython.display import display, Image
from PIL import Image as PILImage

# Load and convert the images
img1 = PILImage.open(r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
↪Practice\anatomy.webp")
img1.save(r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough Practice\anatomy.png",␣
↪"PNG")

img2 = PILImage.open(r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
↪Practice\AnatomyofPlot-1.webp")
img2.save(r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough Practice\AnatomyofPlot-1.
↪png", "PNG")

img3 = PILImage.open(r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
↪Practice\basics_matplotlib.webp")
img3.save(r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
↪Practice\basics_matplotlib.png", "PNG")
```

```
# Display the images
display(Image(filename=r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
 ↪Practice\anatomy.png"))
display(Image(filename=r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
 ↪Practice\AnatomyofPlot-1.png"))
display(Image(filename=r"C:\Users\sonua\OneDrive\Desktop\FSDS\Rough␣
 ↪Practice\basics_matplotlib.png"))
```




Social media vs Official statistics
Highest peak
Proportion of Visitors / SoMe-Users
Month
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
0.12
0.11
0.10
0.09
0.08
0.07
0.06
0.05
0.04
Official Stats
Some users
tick –label (y)
y -label
Title
annotation
line
y –axis
grid lines
filling
tick –label (x)
x –label
ticks
Legend
padding (i.e the distance between label and the axis)
x –axis



1.6.2 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')
```

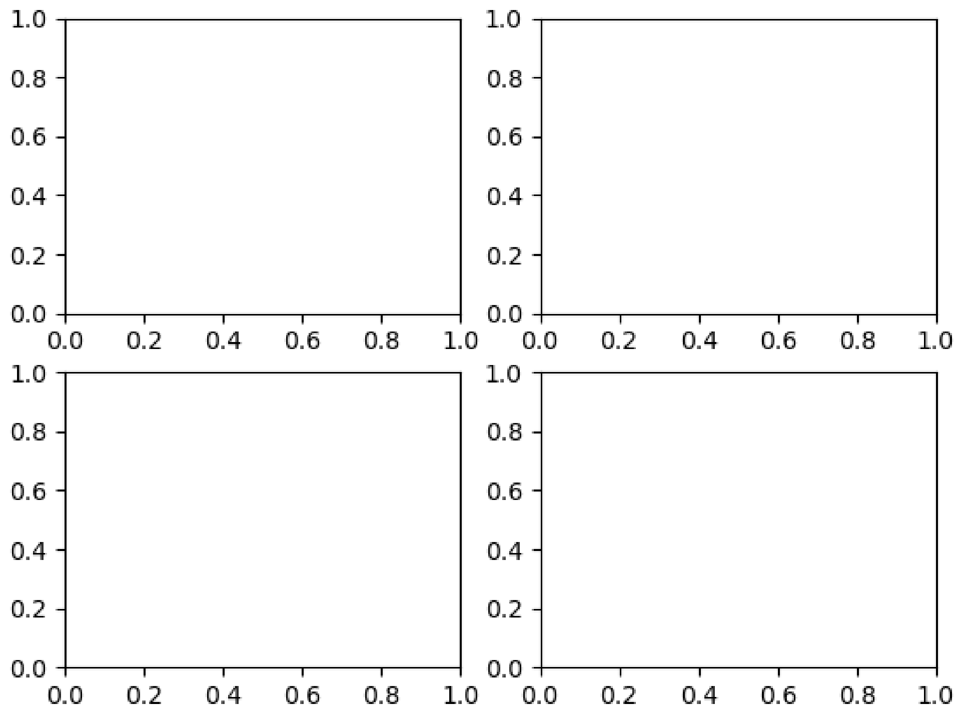
In **savefig()** command, the file format is inferred from the extension of the given filename. Depending on the backend, many different file formats are available. The list of supported file types can be found by using the get_supported_filetypes() method of the figure canvas object as follows:-

```
fig.canvas.get_supported_filetypes()
```

[19]:
```
# Saving the figure

fig.savefig('plot1.png') # save the figure as a png file
```

[20]:
```
# Explore the contents of figure

from IPython.display import Image

Image('plot1.png')
```

[20]:



[21]:
```
# Explore supported file formats


fig.canvas.get_supported_filetypes()
```

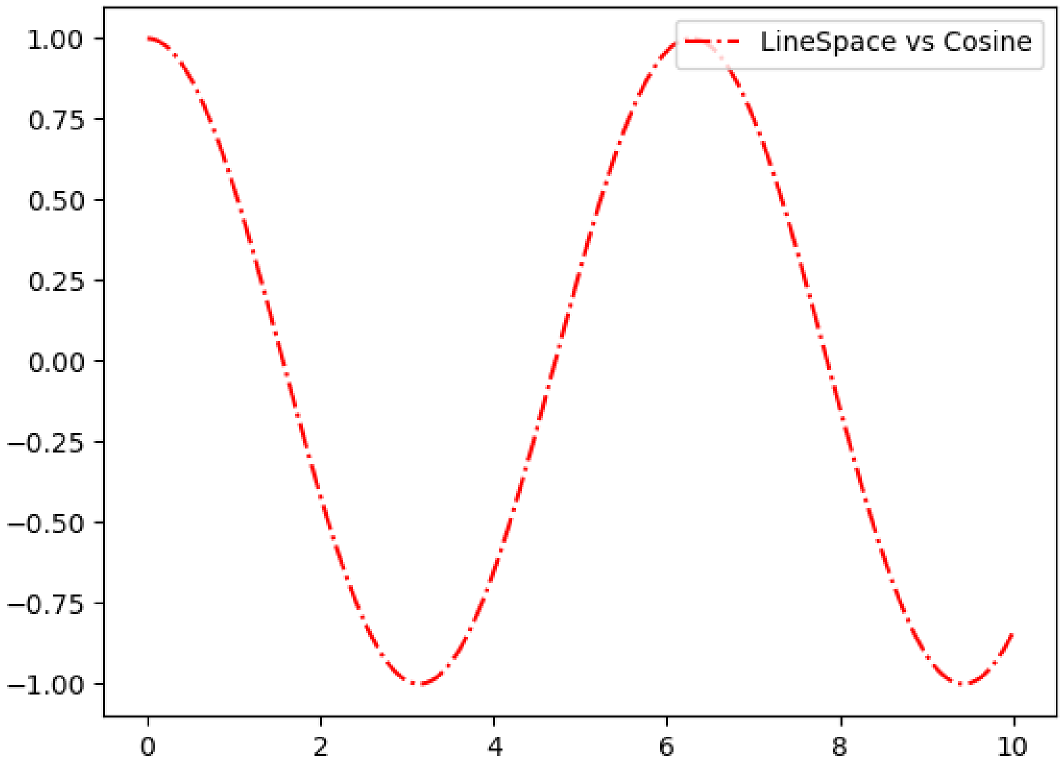
[21]:
```
{'eps': 'Encapsulated Postscript',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'ps': 'Postscript',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format',
 'webp': 'WebP Image Format'}
```

1.7 Line Plot

Commands to draw line plot

[22]:
```
# Create a figure or a container for the plot
fig = plt.figure()

# Create an axes or a plot inside the figure
ax = plt.axes()

# Declare the x and y values
x = np.linspace(0,10,1000)

# Plot the x and y values  and y = cos(x)
ax.plot(x,np.cos(x),'r-.',label='LineSpace vs Cosine')
plt.legend(loc='upper right')
```

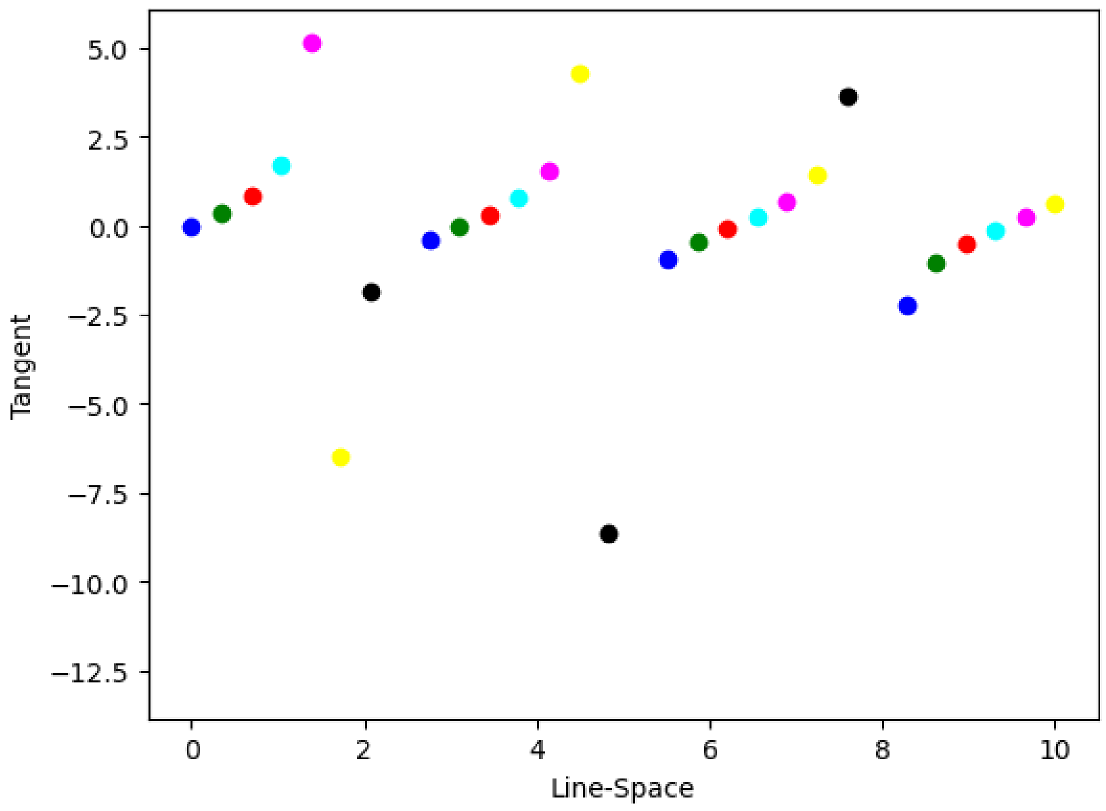
[22]:
```
<matplotlib.legend.Legend at 0x1dfe2d06190>
```



1.8 Scatter Plot

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

[23]:
```
l = np.linspace(0,10,30)
m = np.tan(l)
# list of colors to be used
c =␣
 ↪['blue','green','red','cyan','magenta','yellow','black','white','blue','green','red','cyan'
# loop through the list of colors and plot the points
for i in range(len(l)):
        plt.plot(l[i], m[i], 'o', color=c[i])
plt.xlabel('Line-Space')
plt.ylabel('Tangent')
plt.show()
```

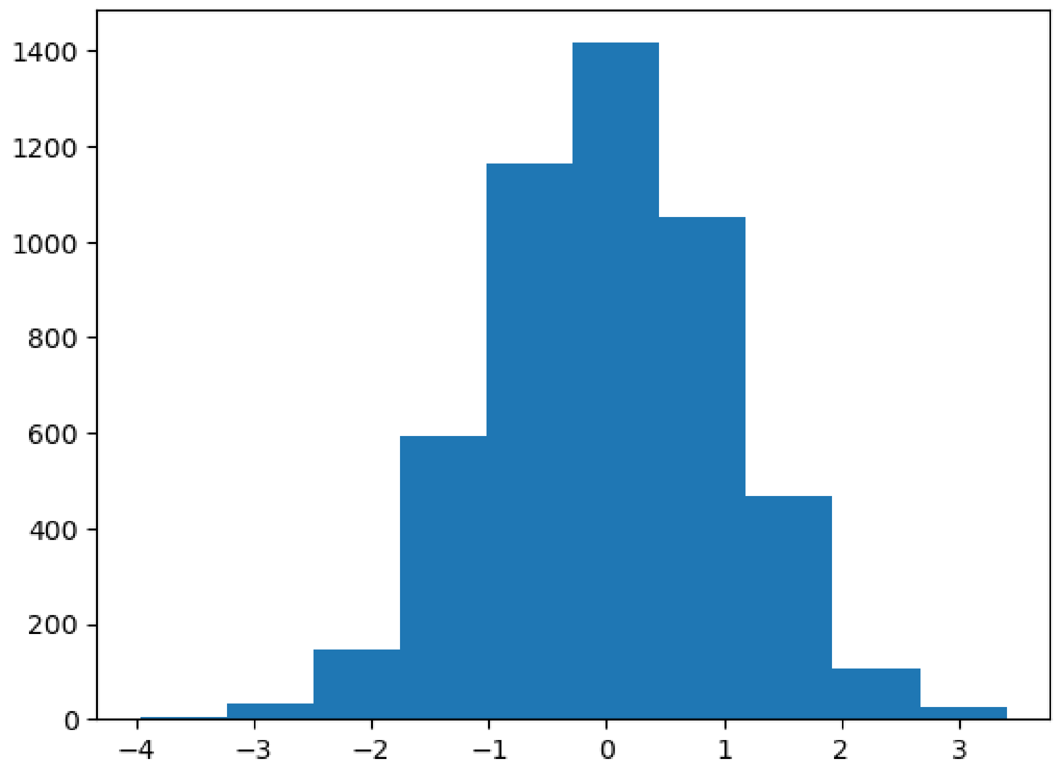


1.9 Histograms

istogram 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:-

[24]:
```
# create a data using random values of 5000
data = np.random.randn(5000)

plt.hist(data)
plt.show()
```

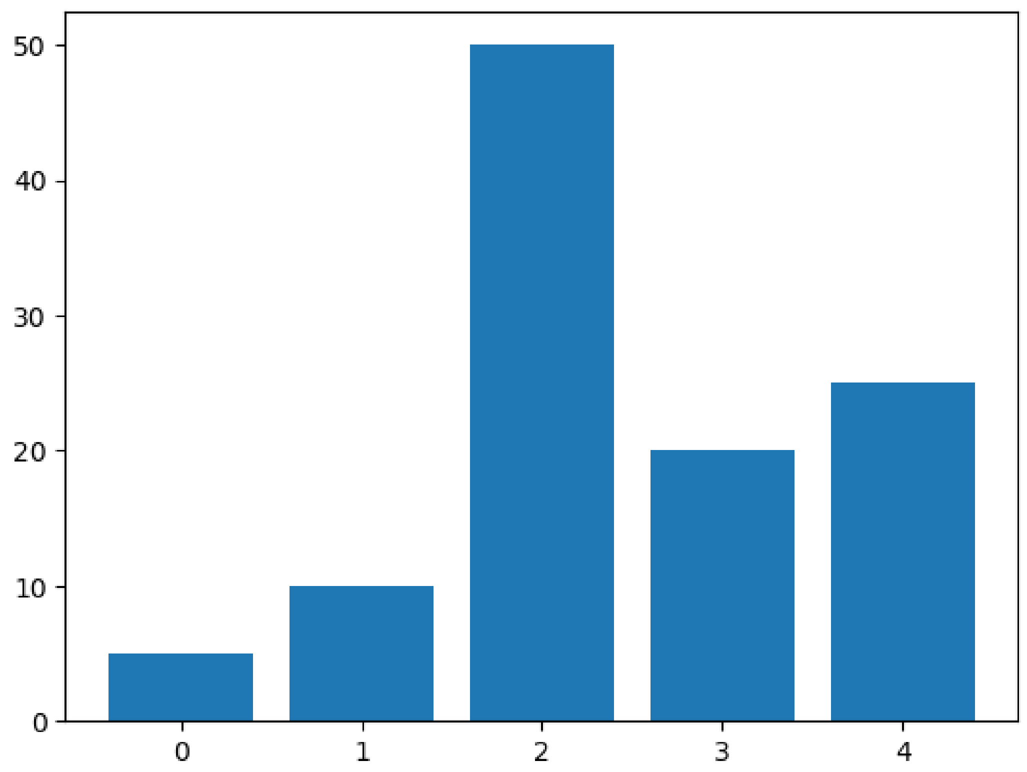


1.10 Bar Chart

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:-

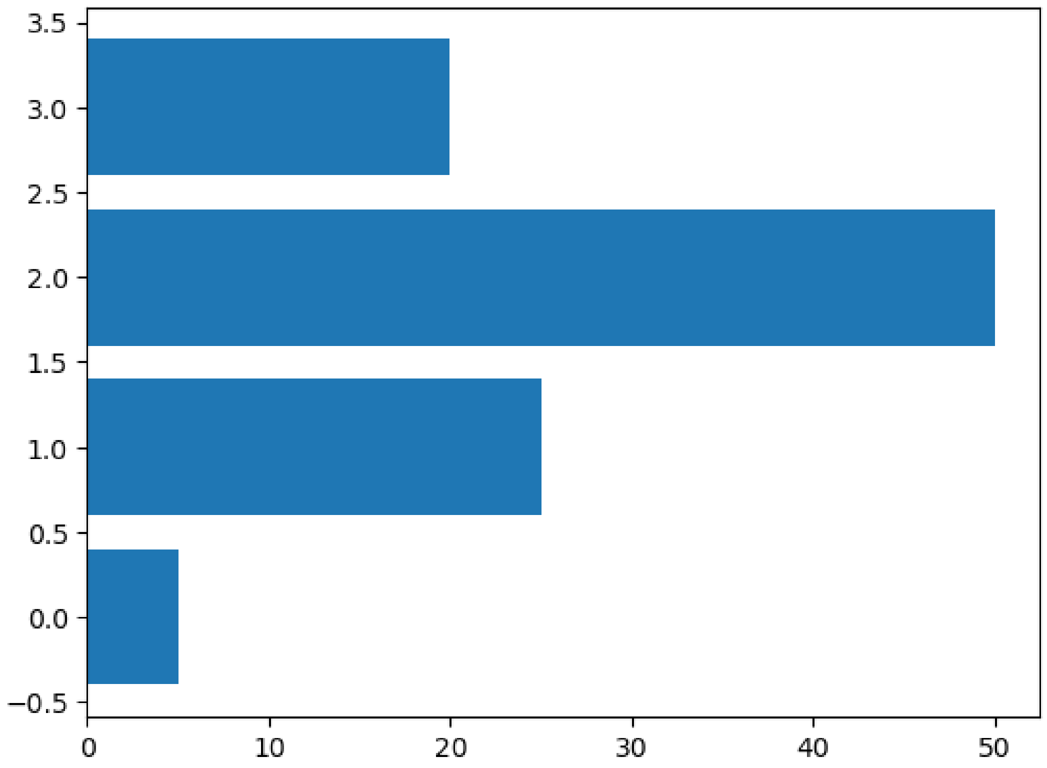
[25]:
```
# create a random data
data2 = [5 , 10, 50, 20, 25]

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



1.11 Horizontal Bar Graph

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

[26]:
```
data2 = [5. , 25. , 50. , 20.]

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

plt.show()
```

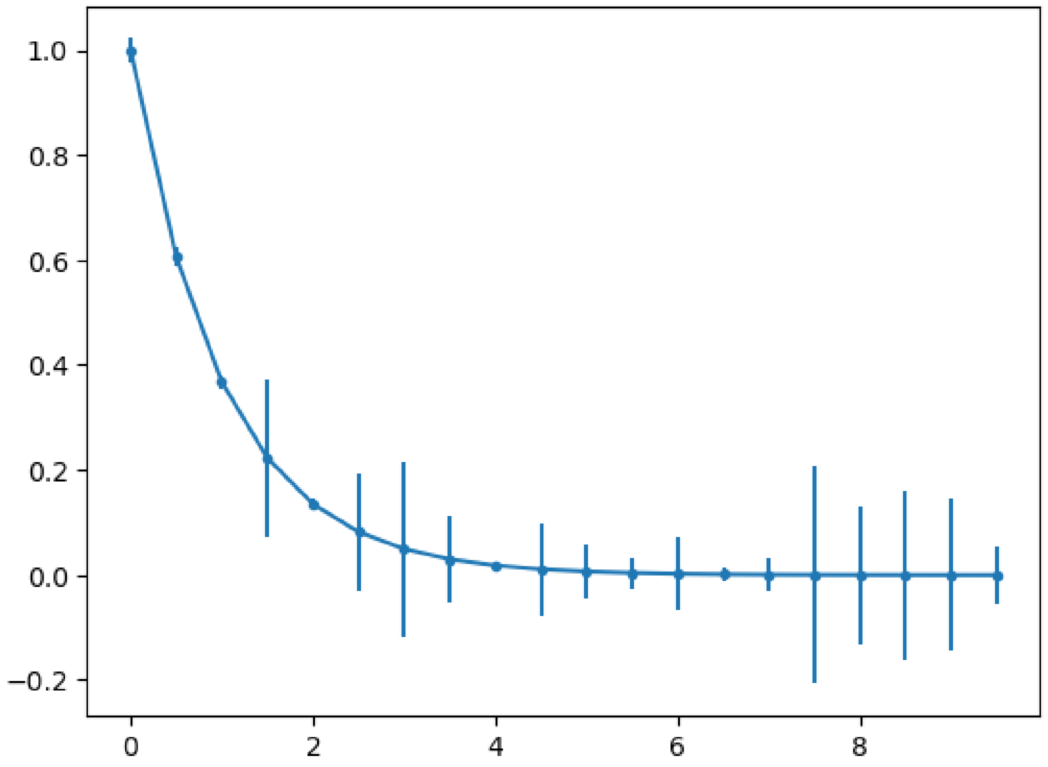


1.12 Error Bar Chart

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:-

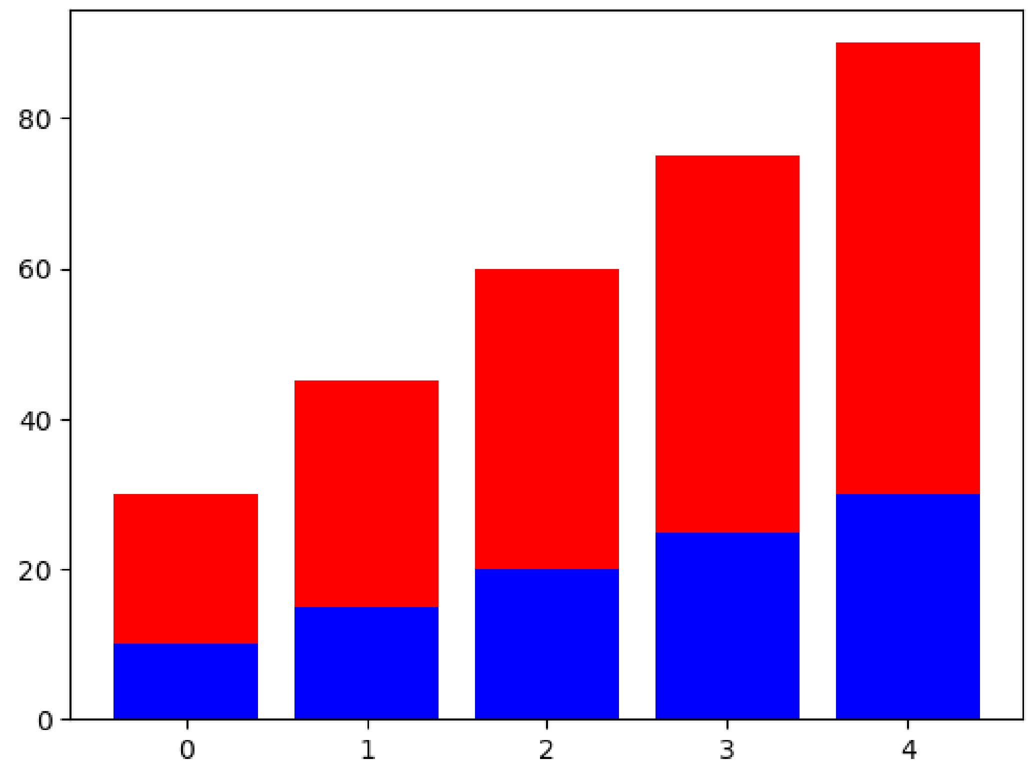
[27]:
```
x5 = np.arange(0,10,0.5)
y5 = np.exp(-x5) # y = e^-x
e1 = 0.1*np.abs(np.random.randn(len(y5))) # error values
plt.errorbar(x5,y5,yerr=e1,fmt='.-') # plot the error bars
plt.show()
```



1.13 Stacked Bar Chart

we can draw stacked bar chart using a special parameter called bottom

[28]:
```
a = [20,30,40,50,60]
b = [10,15,20,25,30]
z2 = range(5)
plt.bar(z2,b,color='b')
plt.bar(z2,a,bottom=b,color='r')
plt.show()
```

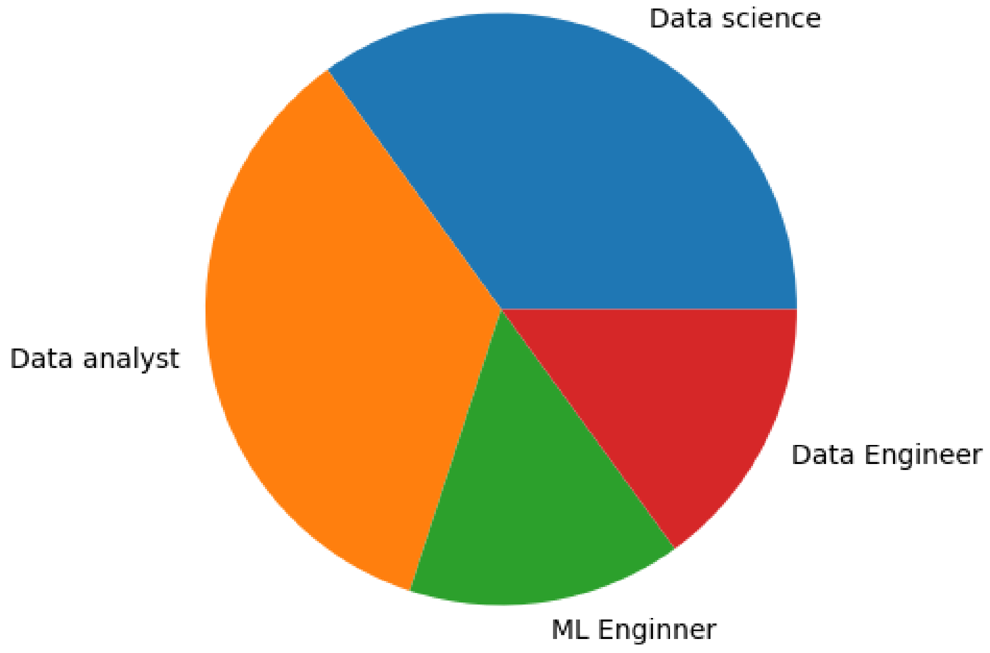


1.14 Pie Chart

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).

[29]:
```
plt.figure(figsize=(10,5))
x11 = [35,35,15,15] # values
# labels
labels = ['Data science','Data analyst','ML Enginner','Data Engineer']
# plot the pie chart
plt.pie(x11,labels=labels)
plt.show()
```

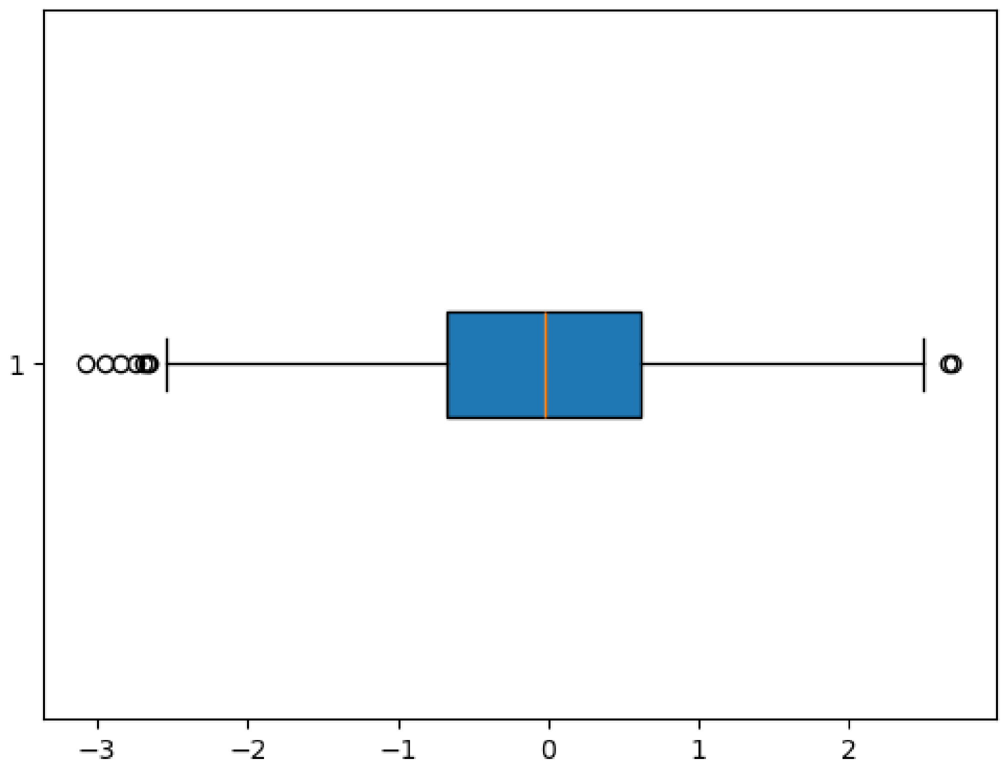


1.15 Box Plot

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:-

[30]:
```python
#Create a data set using numpy
d = np.random.randn(1000)
#plot the box plot
plt.boxplot(d,vert=False,patch_artist=True)
plt.show()
```

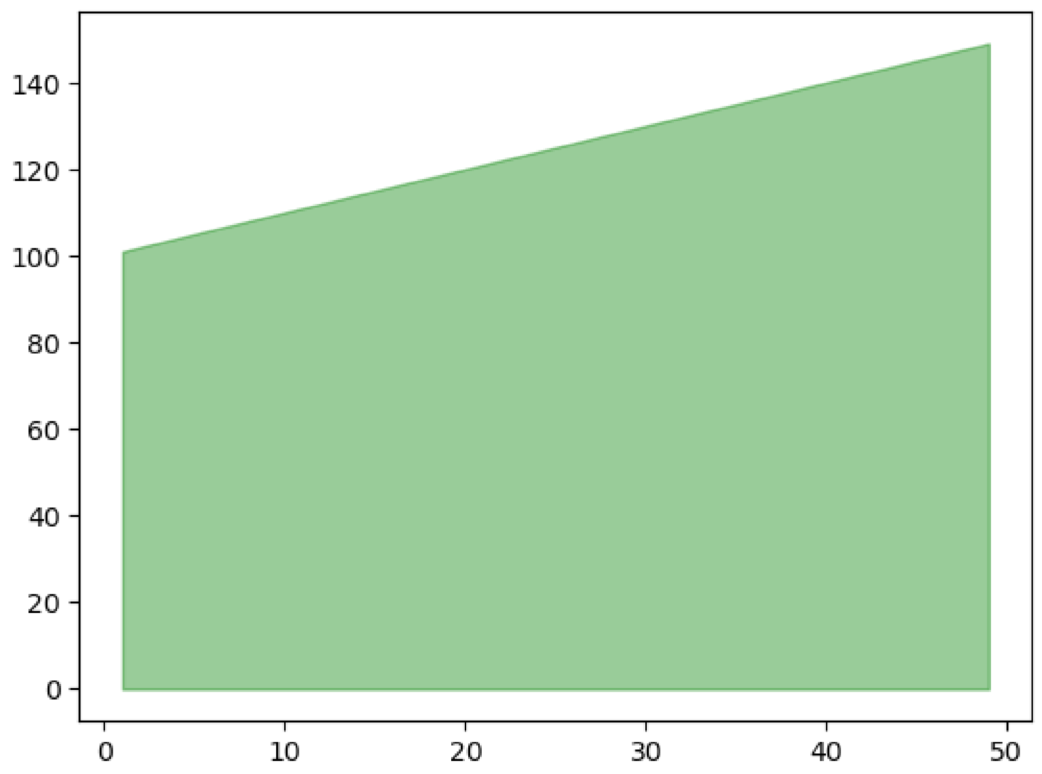


1.16 Area Chart

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:-

[31]:
```
# create a data set using range
data = range(1, 50)
data2 = range(101,150) # Ensure data2 has the same length as data

# Area plot
plt.fill_between(data, data2, color="green", alpha=0.4)
plt.show()
```

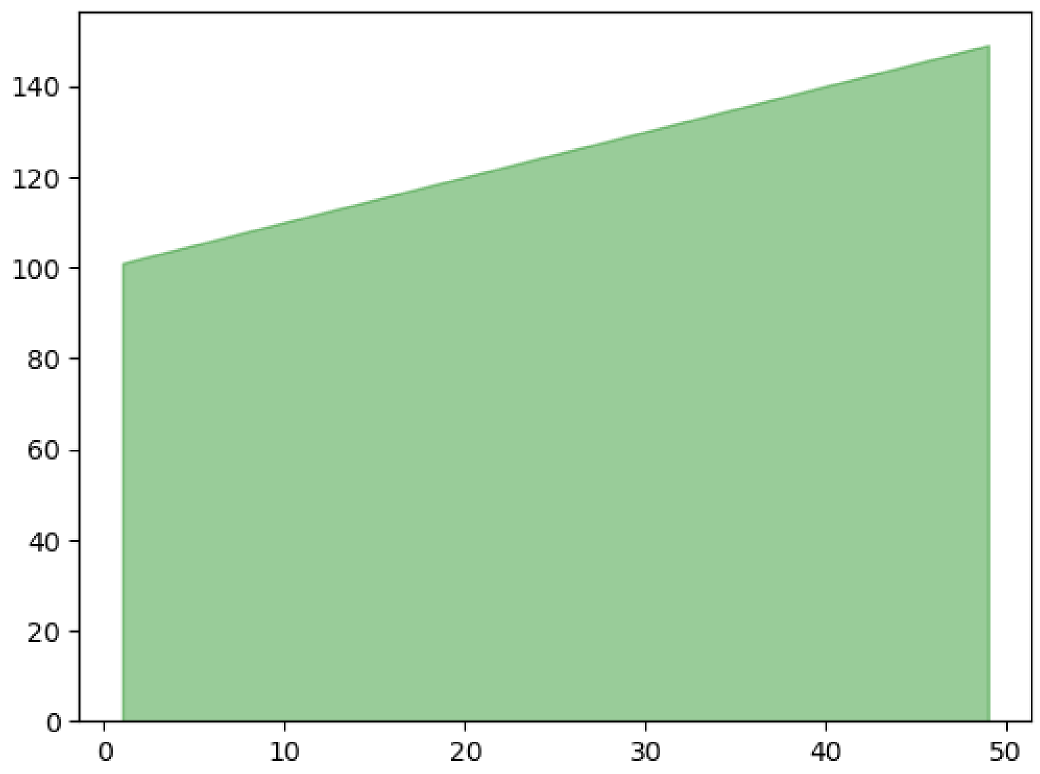


I have created a basic Area chart. I could also use the stackplot function to create the Area chart as follows:-

`plt.stackplot(x12, y12)`

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

[32]:
```
# create a data set using range
data = range(1, 50)
data2 = range(101,150) # Ensure data2 has the same length as data

# Area plot
plt.stackplot(data, data2, color="green", alpha=0.4)
plt.show()
```



1.17 Contour Plot

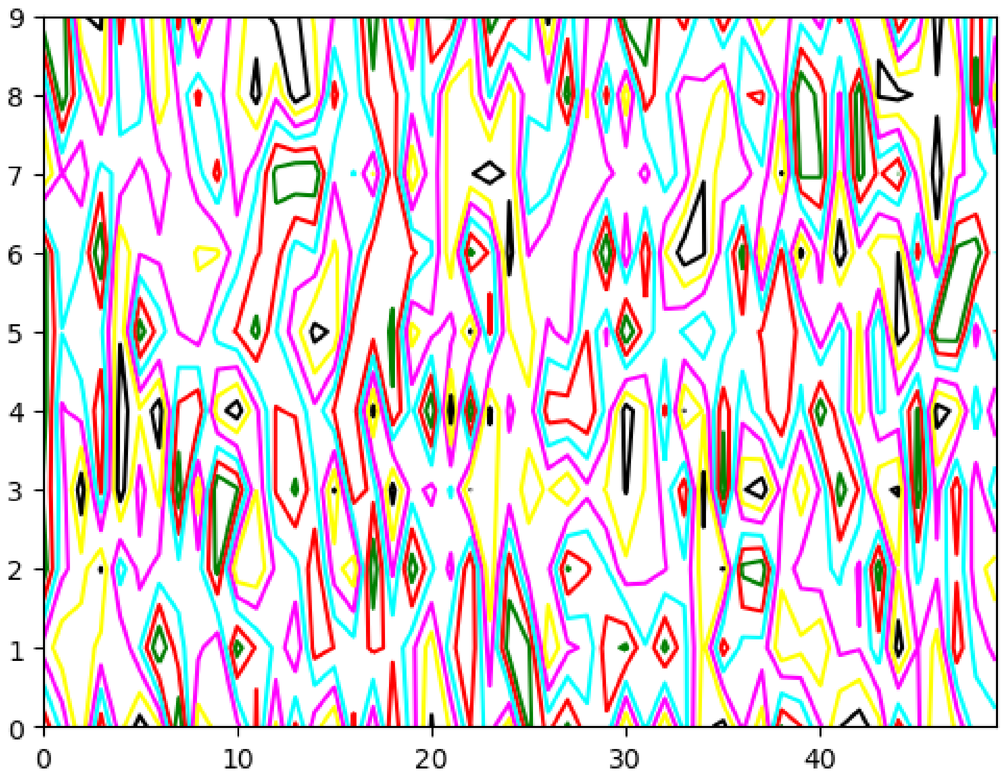
Contour plots are useful to display three-dimensional data in two dimensions using contours or color-coded regions. **Contour lines** are also known as **level lines** or **isolines**. **Contour lines** for a function of two variables are curves where the function has constant values. They have specific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

[33]:
```
# Create a matrix of random values
m= np.random.rand(10,50)
colors =␣
 ↪['blue','green','red','cyan','magenta','yellow','black','white','blue','green']
plt.contour(m, colors=[x for x in colors])
plt.show()
```

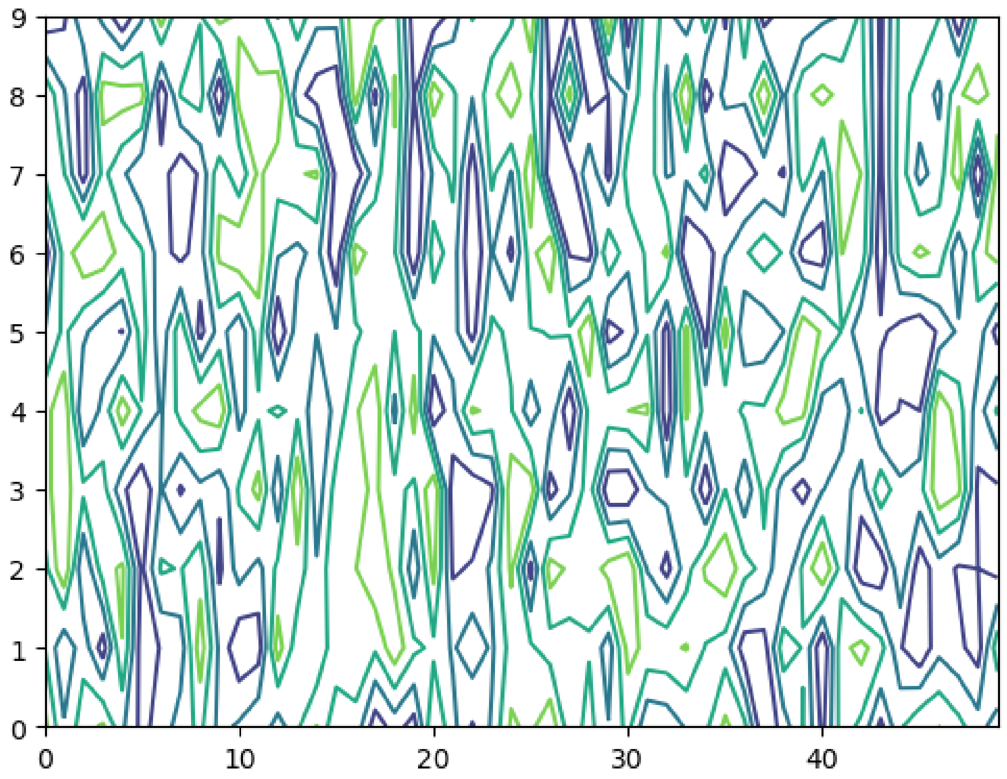


The **contour()** function draws contour lines. It takes a 2D array as input.Here, it is a matrix of 10 x 50 random elements.

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)`

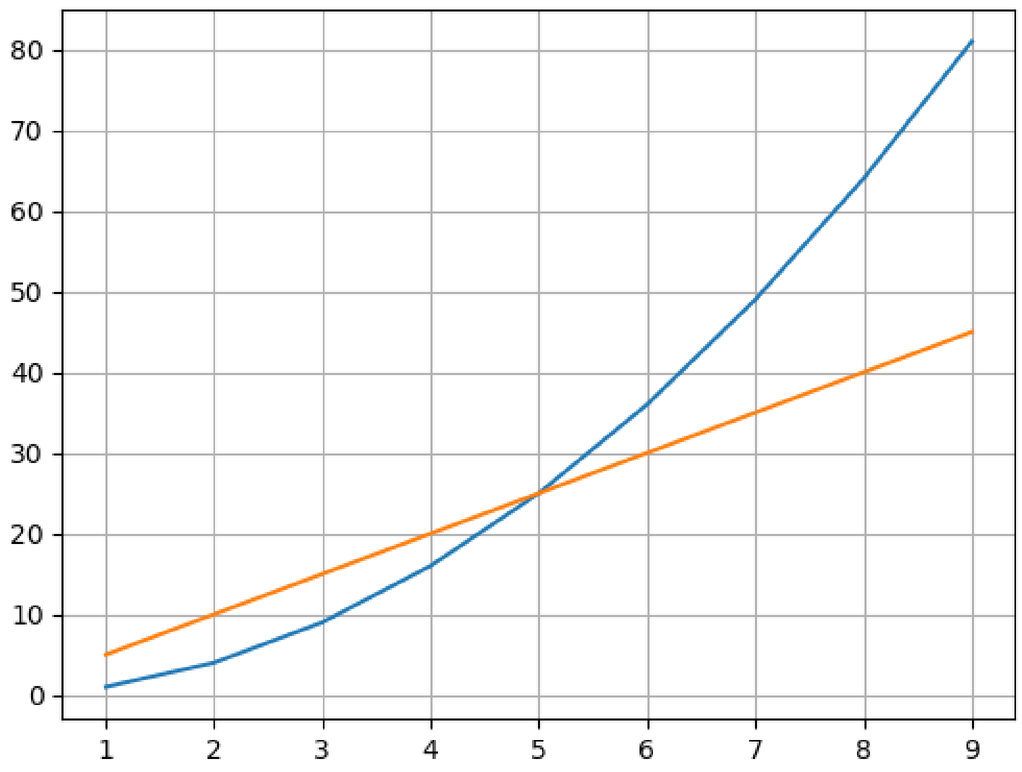
[34]:
```
# Create a matrix of random values
m= np.random.rand(10,50)
plt.contour(m,4)
plt.show()
```



1.18 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.

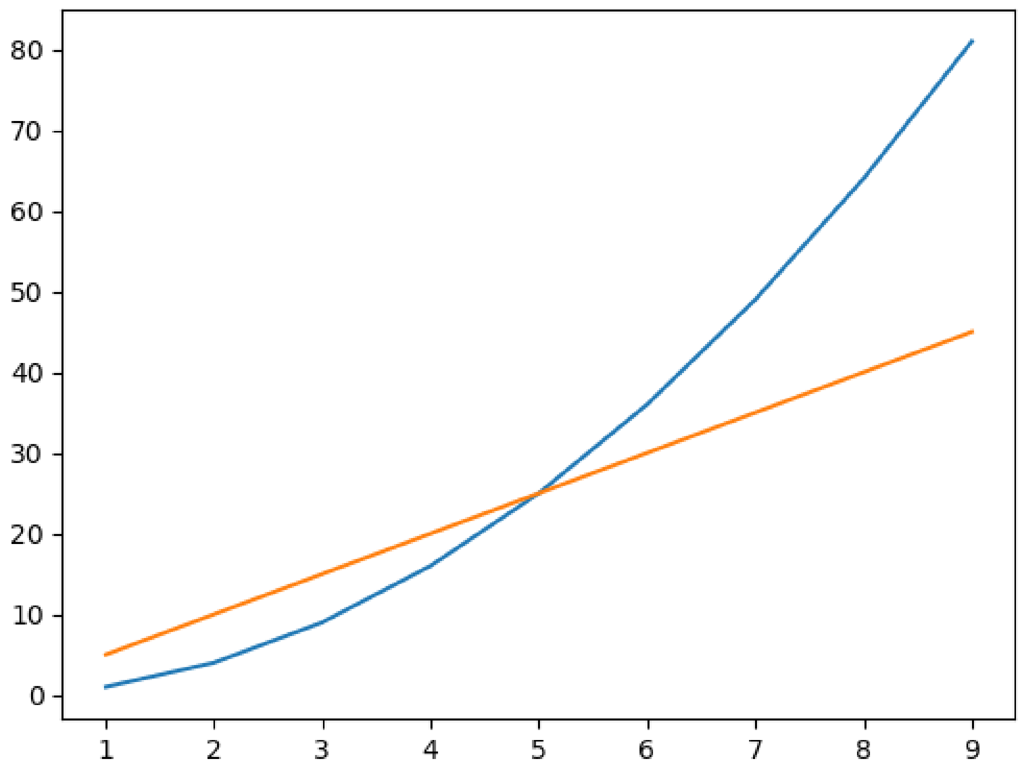
[35]:
```
x11 = np.arange(1,10)
plt.plot(x11, x11**2, x11,x11*5,label='X^2')
# pass true for grid
plt.grid(True)
plt.show()
```



1.19 Handling axes

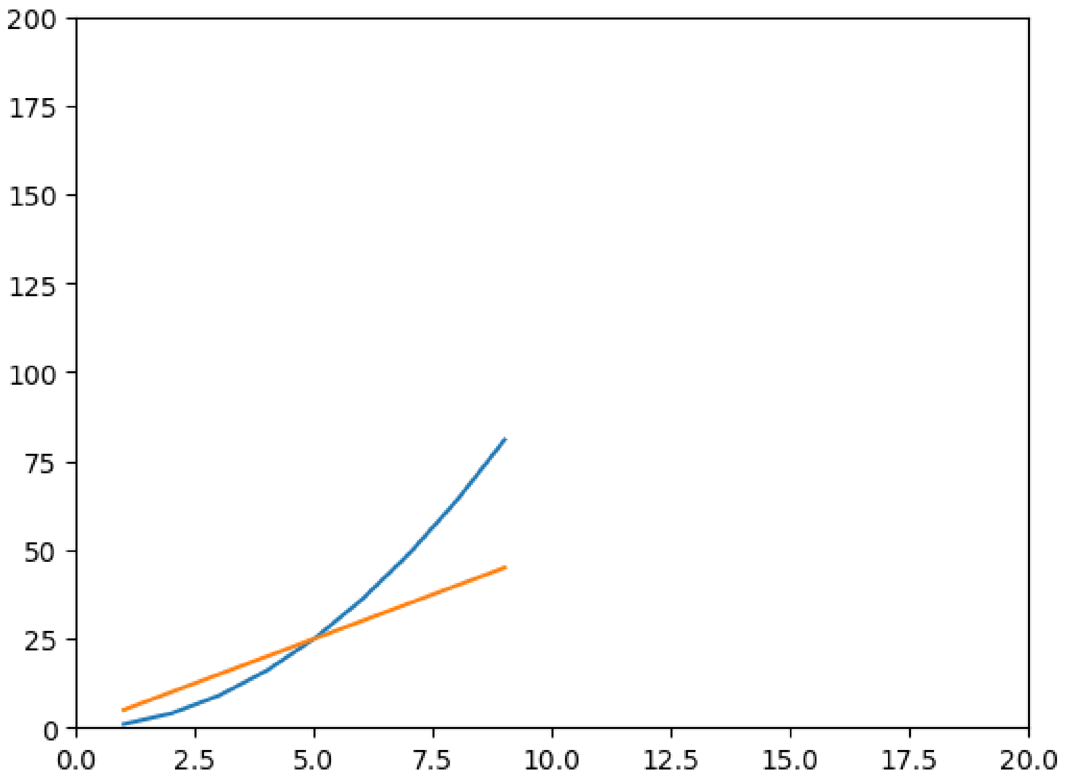
Matplotlib automatically sets the limits of the plot to precisely contain the plotted datasets. Sometimes, we want to set the axes limits ourself. We can set the axes limits with the **axis()** function as follows:-

[38]:
```
x07 = np.arange(1,10)
# Create plots
plt.plot(x07, x07**2, x07,x07*5,label='X^2')

plt.axis()
# Set the axis limits
#plt.axis( [0, 10, 0, 100] )

plt.show()
```



See the diffrence between these two plots

```
[39]: x07 = np.arange(1,10)
      # Create plots
      plt.plot(x07, x07**2, x07,x07*5,label='X^2')

      plt.axis()
      # Set the axis limits
      plt.axis( [0, 20, 0, 200] )

      plt.show()
```



We can see we have more space in the plot now. The x-axis ranges from 0 to 20 and the y-axis ranges from 0 to 200. If we execute **axis()** without parameters, it returns the actual axis limits.

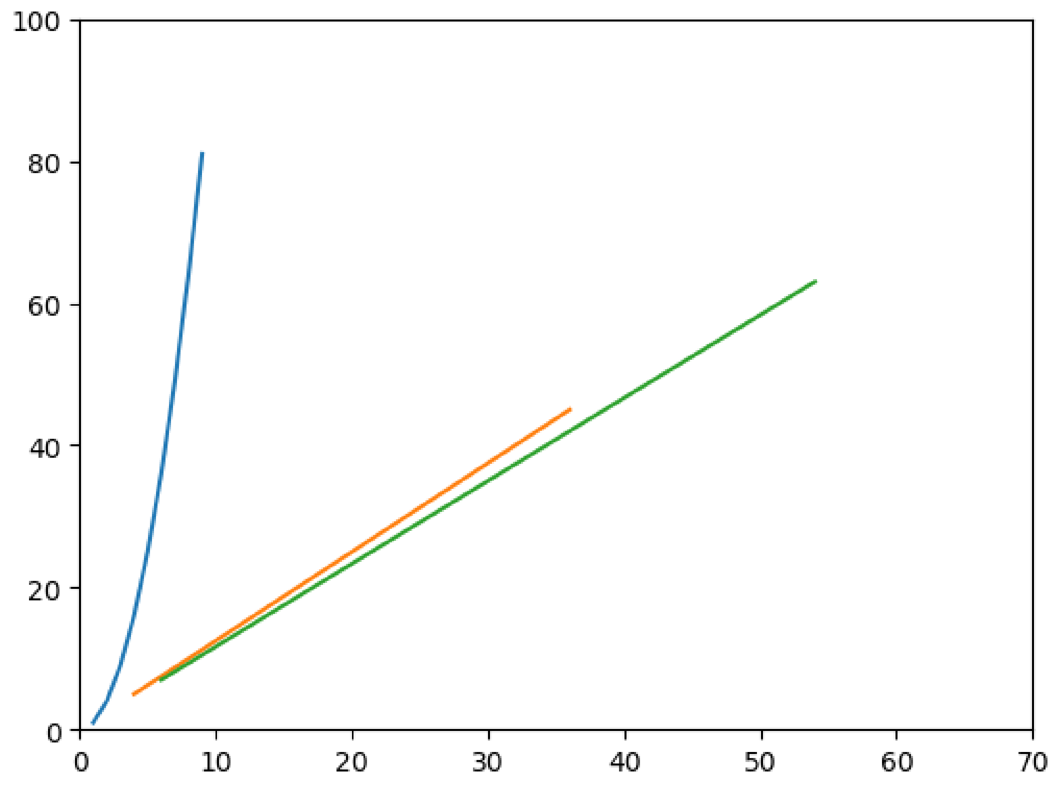
We can set parameters to **axis()** by a list of four values.

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:-

```
[ ]: x07 = np.arange(1,10)
     plt.plot(x07,x07**2,x07*4,x07*5,x07*6,x07*7)
     plt.xlim([0,70])
     plt.ylim([0,100])
```

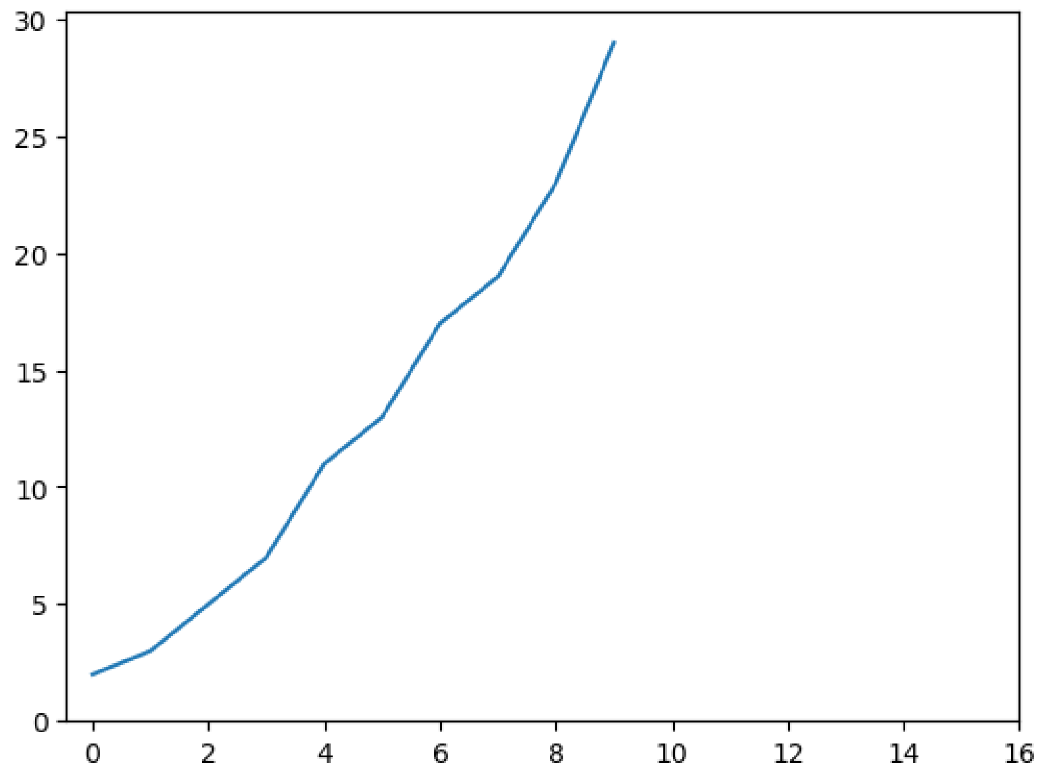
```
[ ]: (0.0, 100.0)
```



1.20 Handling X and Y Ticks

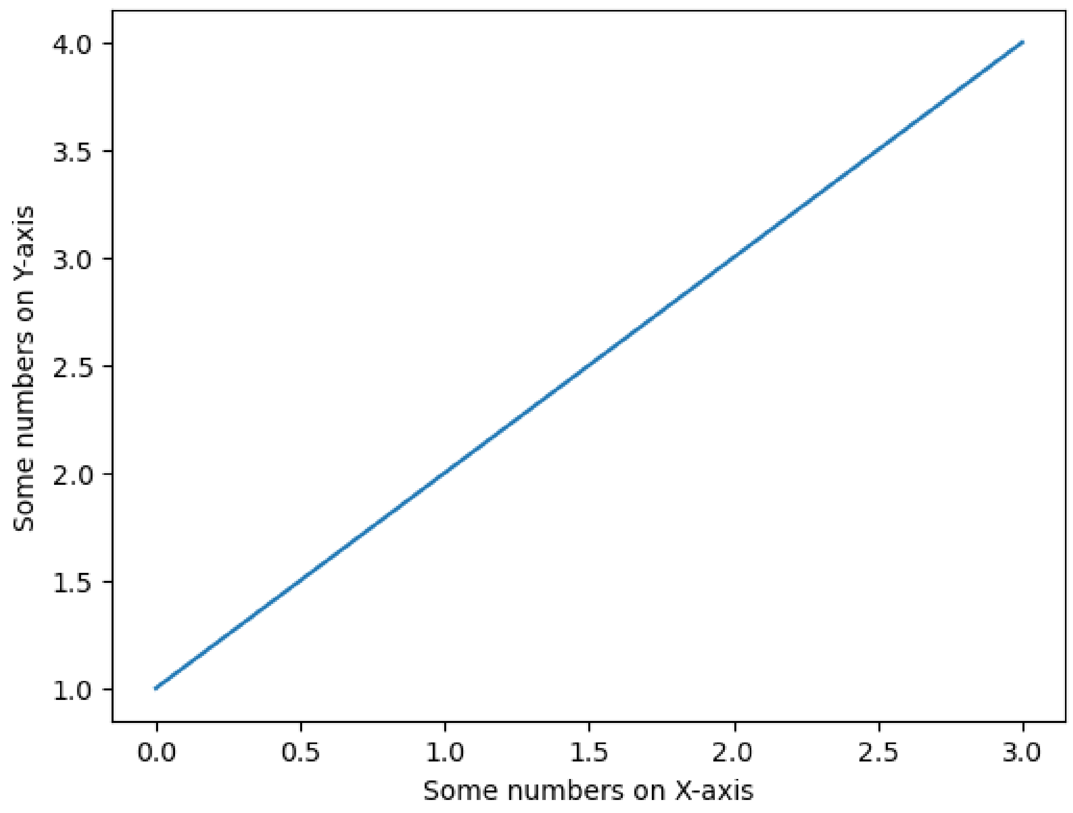
Matplotlib provides two basic functions to manage them - **xticks()** and **yticks()**.

[49]:
```
x = [2,3,5,7,11,13,17,19,23,29]
plt.plot(x)
plt.xticks([0,2,4,6,8,10,12,14,16])
plt.yticks([0,5,10,15,20,25,30])
plt.show()
```



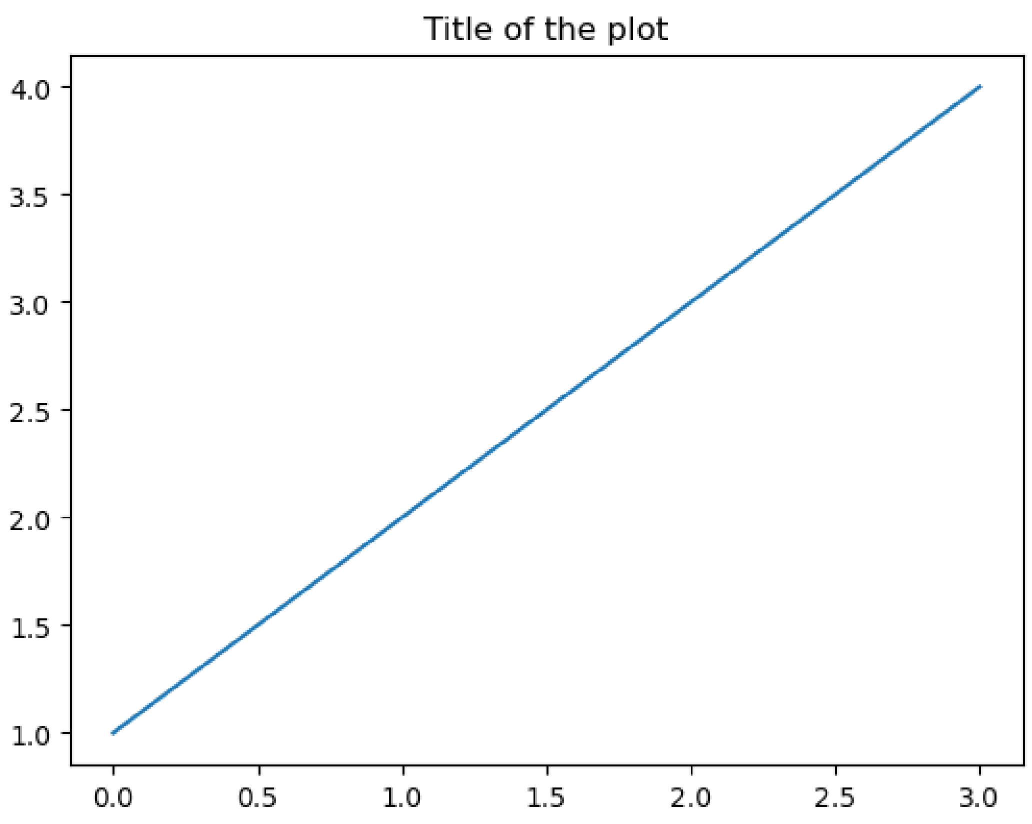
1.21 Adding Labels

[50]:
```
plt.plot([1,2,3,4])
plt.xlabel('Some numbers on X-axis')
plt.ylabel('Some numbers on Y-axis')
plt.show()
```



1.22 Adding a title

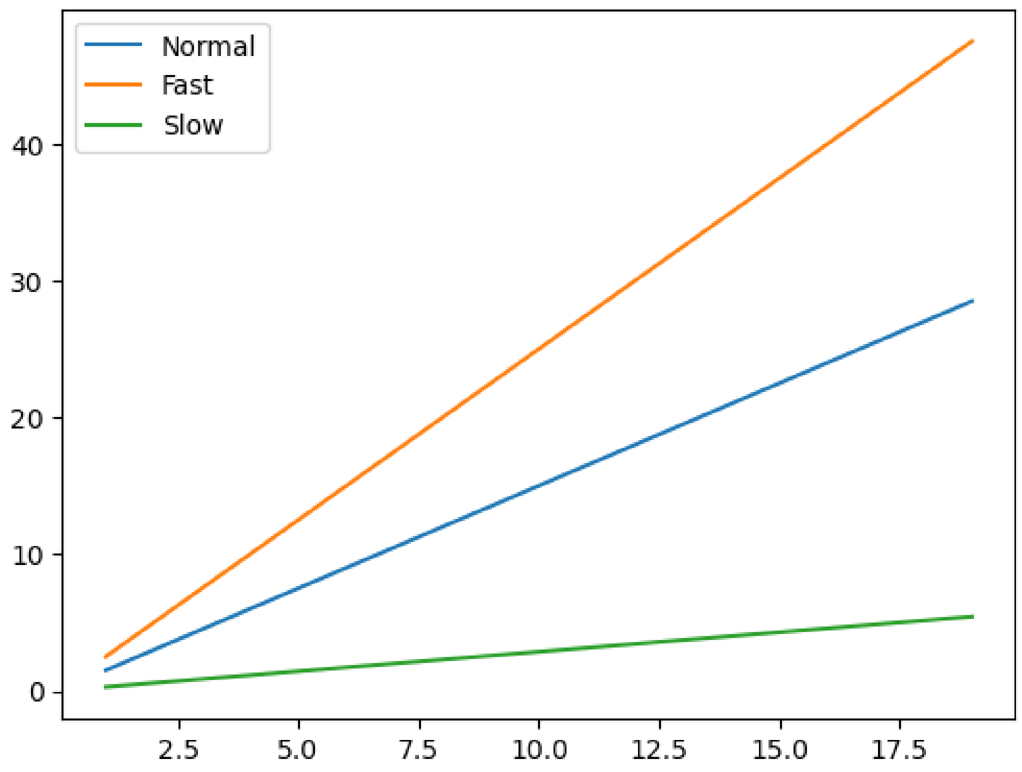
```
[ ]: plt.plot([1,2,3,4])
     plt.title('Title of the plot') # Title of the plot
     plt.show()
```



1.23 Adding a legend

[ ]:
```
x15 = np.arange(1, 20)

fig, ax = plt.subplots()

ax.plot(x15, x15*1.5)
ax.plot(x15, x15*2.5)
ax.plot(x15, x15/3.5)

ax.legend(['Normal','Fast','Slow']) # Add a legend
```

[ ]:
```
<matplotlib.legend.Legend at 0x1dfea3fc450>
```



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

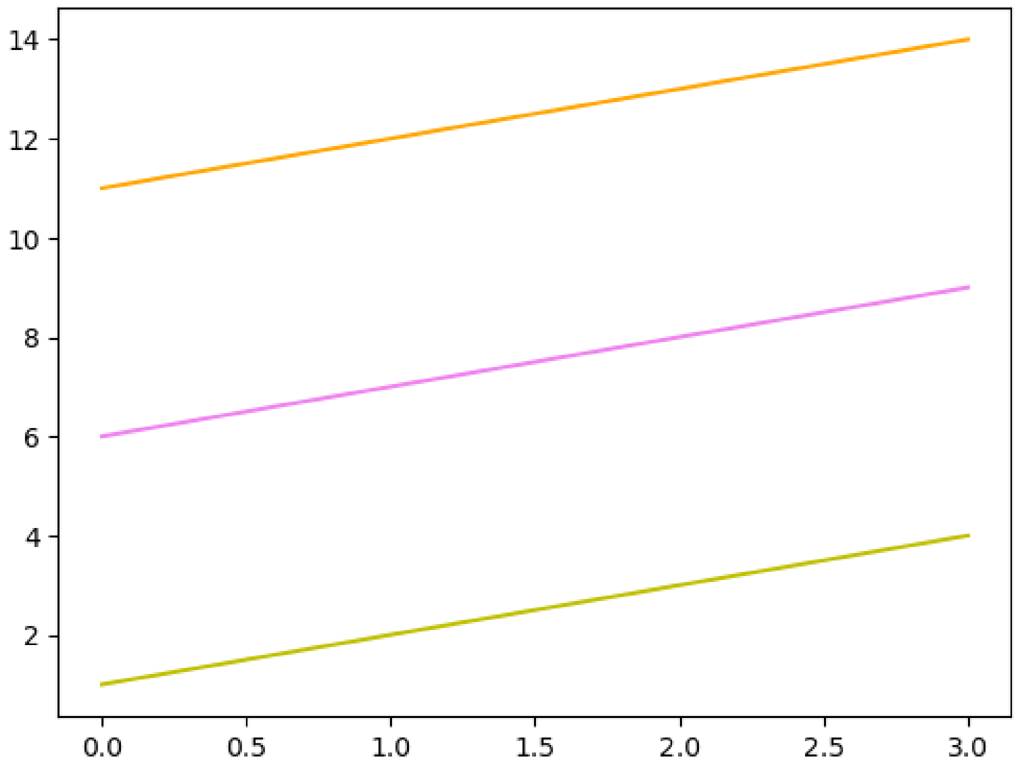
ax.legend(loc=8) # lower center

ax.legend(loc=9) # upper center

ax.legend(loc=10) # center

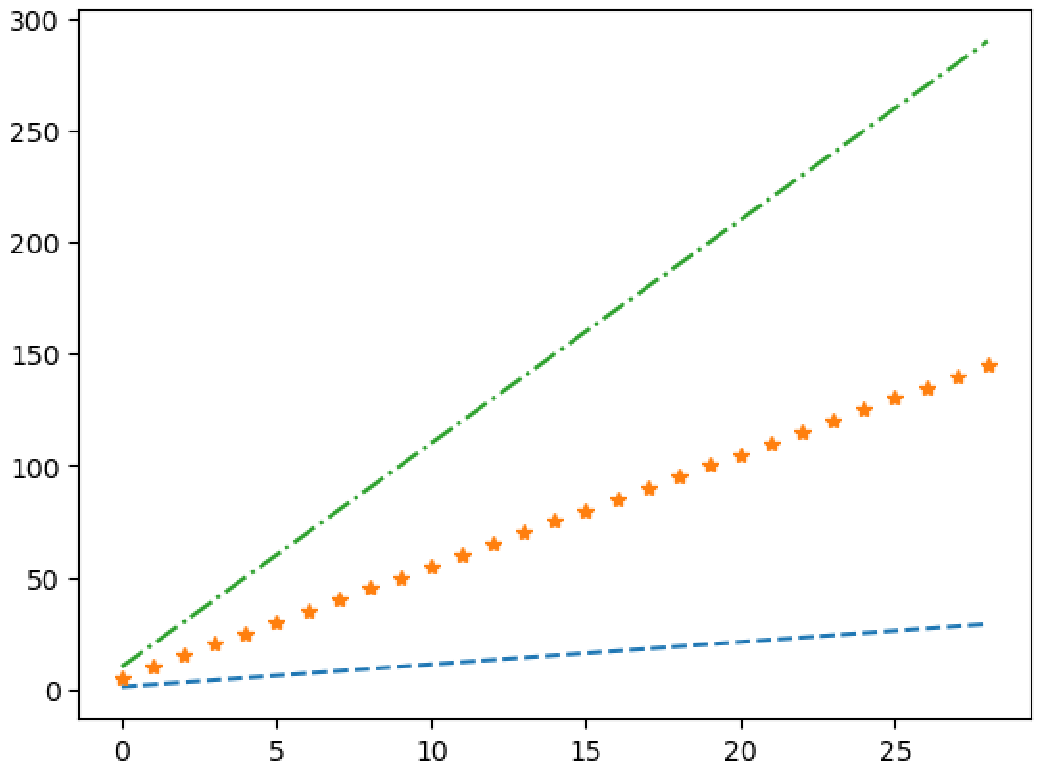
1.24 Control Colours

[57]:
```
x16 = np.arange(1, 5)

plt.plot(x16, 'y')
plt.plot(x16+5, 'violet')
plt.plot(x16+10, 'orange')

plt.show()
```



1.25 Control Lines

[ ]:
```
x= np.arange(1,30)
plt.plot(x,'--',x*5,'*',x*10,'-.')
plt.show()
```



1.26 Summary

In this code snippet, we have learned how to create different types of plots using the Matplotlib library in Python. We have learned how to create a simple plot, a plot with multiple lines, a plot with multiple plots, a plot with multiple panels, a plot with multiple axes, a plot with multiple figures, a plot with multiple subplots, a plot with multiple containers, a plot with multiple axes, a plot with multiple plots, a plot with multiple lines, a plot with multiple colors, a plot with multiple markers, a plot with multiple labels, a plot with multiple legends, a plot with multiple titles a plot with multiple labels 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