**Matplotlib:**

Matplotlib library for python is a visualization library for creating many types of graphs like static graphs, animated graphs and interactive visualizations. Most of the Matplotlib utilities lies under the pyplot submodule, and are imported under the plt alias.

***“import matplotlib.pyplot as plt”***

Pyplot is an API for python’s maplotlib that effectively makes matplotlib a viable open source alternative MATLAB. MATLAB is a propriety multi-paradigm programming language and numeric computing environment, which can perform data analysis, graphics, algorithm development, app building, etc. MATLAB can be used in python. Pyplot provides matplotlib with two main features:

* A MATLAB style interface, which allows the users who are familiar with MATLAB to adopt to it easily.
* Statefulness, which means thath Pyplot stores the state of an object when you plot it. This is essential for use in the same loop until plt.close() is encountered in the code.

Matplotlib needs to be installed in the system before everything. It can be installed many ways. The most simplest way is executing the following code in the command prompt.

“***pip3 install matplotlib***”

It installs the latest version of matplotlib. Matplotlib contains various types of graphs. For example, Line graph, Bar graphs, markers, scatter plot, Pie charts, polar charts, animations, scales, 3D plotting, etc.

Matplotlib makes easy things easier and hard things possible. We can Create, Customize and   
Extend our graphs. We can:

* Develop quality plots with just a few lines of code.
* Use interactive figures that can zoom, pan
* Take full control of line styles, font properties, axes properties
* Export to a number of types of file formats.
* Explore tailored functionality provided by third party packages.

**Getting Started with Matplotlib:**

**A sample example:**

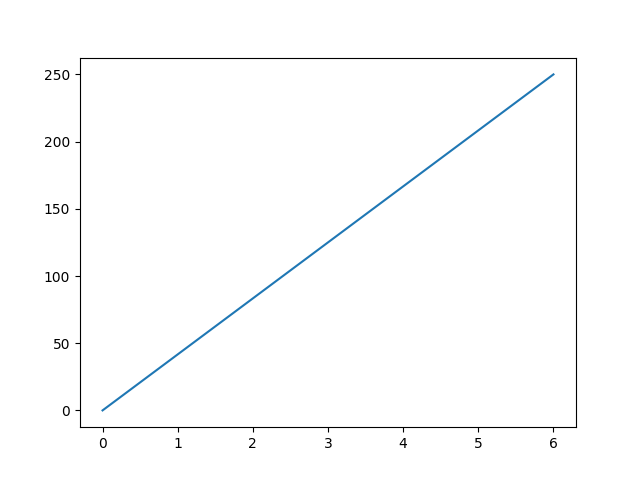
Matplotlib plots the data on Figures. Figures are the windows which contain one or more axes (an area where points are specified in terms of x-y coordinates). A simplest graph is created using the code below.

“import matplotlib.pyplot as plt

import numpy as np

xpoints = np.array([0, 6])  
 ypoints = np.array([0, 250])  
 plt.plot(xpoints, ypoints)  
 plt.show()”

First, the required modules are to be imported which is matplotlib.pyplot and numpy. Numpy is a module used for various matrix operations. Two lists “xpoints” and “ypoints” are created using Numpy module with values ranging from 0-6 and 0-256. This two list values are then plotted in an empty figure with xpoints list as x-coordinates and ypoints list as y-coordinates. Then plt.show() function is called. This function shows the plotted graph in the figure in a new window. The output graph of the above code looks similar to the following:



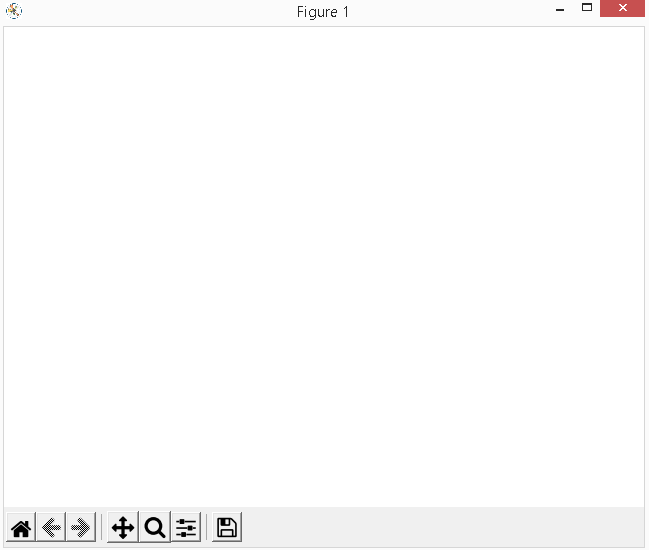
Thus the simple graph is plotted with some random values of x and y coordinates.

The simplest way of creating a figure with an axes is using pyplot.subplots. This pyplot.subplot creates subplots with required number of axes. Lets create an empty figure using “plt.figure()” function.

**Program:**

“import matplotlib.pyplot as plt

Fig = plt.figure()

plt.show()” 

This is the empty figure we have created. Now we can add axes to the empty figure.

**Code:**

“import matplotlib.pyplot as plt

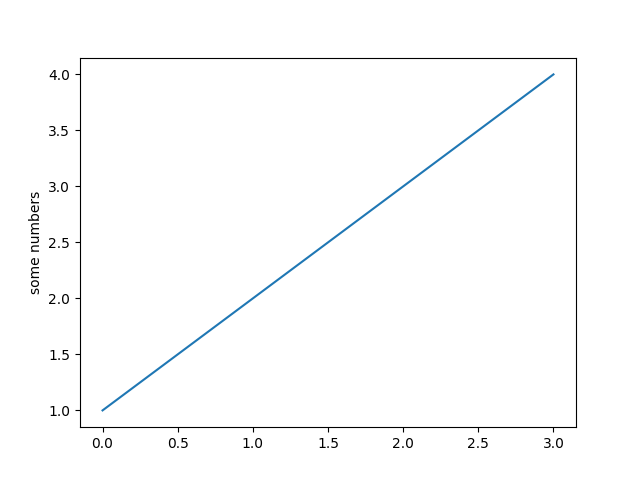
Fig = plt.figure()

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

plt.ylabel('some numbers')

plt.show()”

**Output:**



In the above code we used plt.plot() to plot a list of 4 values. The x-axis ranges from 0-3 and the y-axis ranges from 1- 4. If you provide a single list of values to plot, matplotlib assumes it as a sequence of y-values and automatically generates values for x. Since python range generally starts form 0, the default value of x axis starts from 0, and it has the same length as y. So the x-values generated are [0,1,2,3]. The plt.ylabel() function is used to set the label for the y axis. Similarly, plt.xlabel() function for x-axis. The x-axis and y-axis values limit can also be given so that the output graph can be plotted in a given range. It can be achieved by using plt.xlim() and plt.ylim(). A title can be given to a figure using plt.title() function.

**Code:**

“Fig = plt.figure()

plt.title("Sample Plot")

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

plt.ylabel('some numbers')

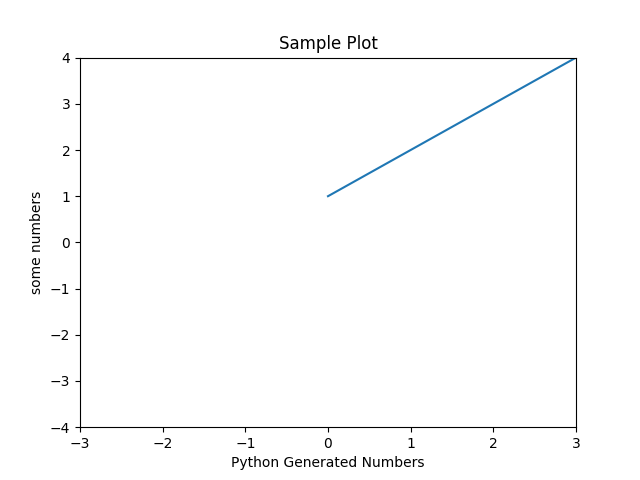
plt.xlabel("Python Generated Numbers")

plt.xlim(-3,3)

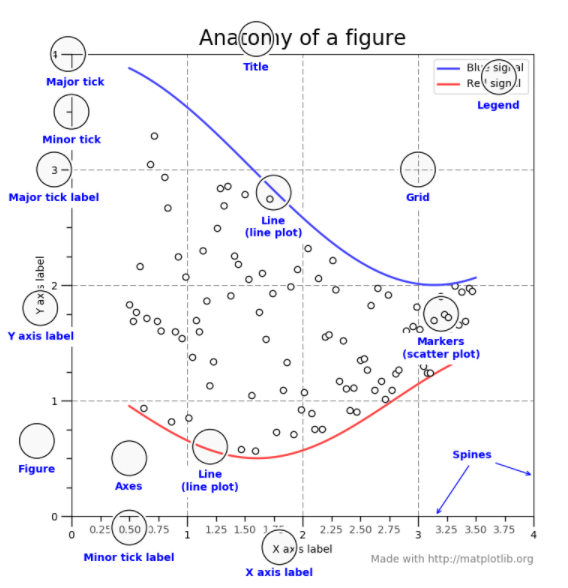
plt.ylim(-4,4)

plt.show()”

**Output:**



This is the output of the above code. The x and y axis labels are set. The range of x and y axis are given using xlim(), ylim function. Since the range is from negative to positive number and the values are starting from 0, the graph is plotted accordingly.

**Parts of a Figure:**

**Multiple Subplots in a figure:**

An entire figure can be divided into two or more subplots. Maplotlib makes it easier by using plt.subplot(). Subplots can also be generated using axes function in pyolot from matplotlib.

**Example:**

“import numpy as np

import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)

x2 = np.linspace(0.0, 2.0)

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

y2 = np.cos(2 \* np.pi \* x2)

plt.subplot(2, 1, 1)

plt.plot(x1, y1, 'o-')

plt.title('A tale of 2 subplots')

plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)

plt.plot(x2, y2, '.-')

plt.xlabel('time (s)')

plt.ylabel('Undamped')

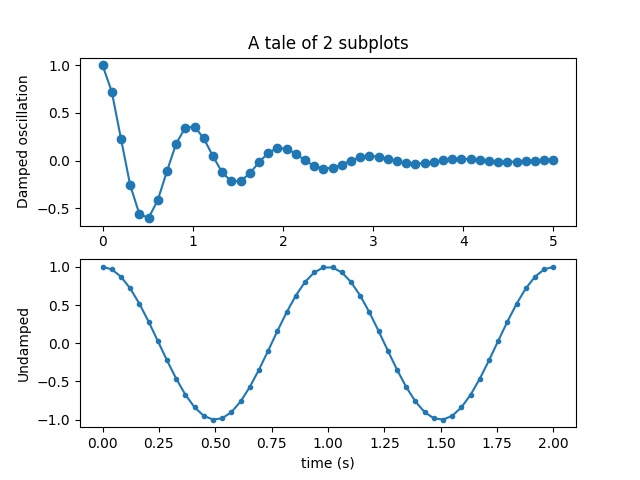
plt.show()”

We are going to plot 2 subplots in a single figure. So 4 list values are required plot the graphs. The pyplot get the values in the form of list and plots it accordingly. X1, x2, y1, y2 are the lists created with some values generated using numpy array module. Then the plt.subplot(2,1,1)

**Plt.subplot():**

It has 3 major parameters. Plt.subplot(int,int,int). plt.subplot(1,1,1) is the default. Theree integers are (nrows, ncols, index). The subplot will take the index position on a grid with nrows rows and ncols columns. Index starts at 1 in the upper left corner and increases to the right.

In this subplot(2,1,1) defines 2 rows which means 2 separate plots up and down are plotted separately with same 1 column. It has a common figure title and different y lables.

**Output:** 

**Alternative Method for Creating Multiple Plots:**

Subplots can also be generated by defining the two axes separately.

**Example:**

“import numpy as np

import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)

x2 = np.linspace(0.0, 2.0)

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

y2 = np.cos(2 \* np.pi \* x2)

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

fig.suptitle('A tale of 2 subplots')

ax1.plot(x1, y1, 'o-')

ax1.set\_ylabel('Damped oscillation')

ax2.plot(x2, y2, '.-')

ax2.set\_xlabel('time (s)')

ax2.set\_ylabel('Undamped')

plt.show()”

Its output is same as the previous method. The change is that we used plt.subplot, a pyplot inbuilt function to create multiple subplots. On the other hand, here we created an empty figure. Then two axes ax1,ax2 are created using plt.subplots. Then, the plotting of the axes ax1, ax2 are done separately and the other actions like setting label, axis limit, etc is done. Finally the graph is shown using plt.show().

**Bar Graphs:**

Bar graphs are best used when showing comparisons between categories. Typically, the bars are proportional to the values they represent and can be plotted either horizontally or vertically. One axis of the chart shows the specific categories being compared, and the other axis represents discrete values.

There are 3 main A bar graph is a graph that presents the categorical data with

Rectangular bars with heights or lengths proportional to the values they represent. In python,

matplotlib allows us to create 3 types of bar graphs.

* Grouped Bar Graphs
* Stacked Bar Graphs
* Horizontal Bar Graphs

**Grouped Bar Graphs:**

Grouped bar charts are Bar charts in which multiple sets of data items are compared, with a single color used to denote a specific series across all sets. As with basic Bar charts, both vertical and horizontal versions of grouped bar charts are available. Lets see an example for a vertical grouped bar graph.

**Code:**

“import matplotlib.pyplot as plt

import numpy as np

labels = ['G1', 'G2', 'G3', 'G4', 'G5']

men\_means = [20, 34, 30, 35, 27]

women\_means = [25, 32, 34, 20, 25]

x = np.arange(len(labels)) # the label locations

width = 0.35 # the width of the bars

fig, ax = plt.subplots()

rects1 = ax.bar(x - width/2, men\_means, width, label='Men')

rects2 = ax.bar(x + width/2, women\_means, width, label='Women')

# Add some text for labels, title and custom x-axis tick labels, etc.

ax.set\_ylabel('Scores')

ax.set\_title('Scores by group and gender')

ax.set\_xticks(x)

ax.set\_xticklabels(labels)

ax.legend()

ax.bar\_label(rects1, padding=3)

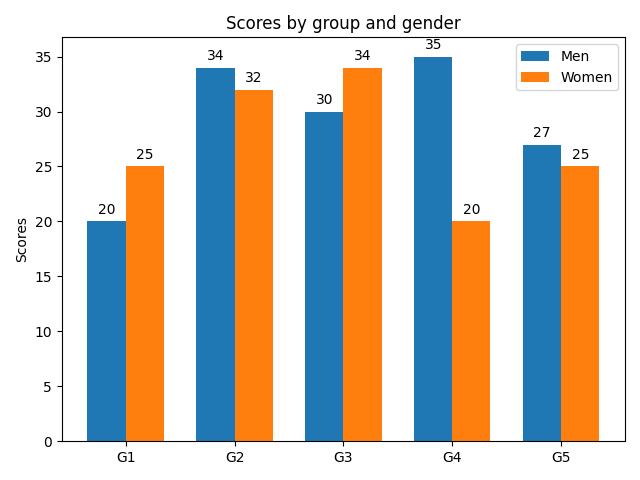
ax.bar\_label(rects2, padding=3)

fig.tight\_layout()

plt.show()”

The required modules are imported like every other program. Three lists labels, men\_means, women\_means with equal number of elements are created. Grouped Bar graph is used to represent a comparison between two or more elements. Here the comparison made is between men\_means and women\_means list. An emptly figure is created using subplots with ax as axes. Since we are comparing two lists here, rects1 and rects2 are the variables that is going to represent the data in the men\_means and women\_means list in the graph. After plotting the data, the axes and the figure should be presented in a professional manner. So x and y axis labels and limits are mentioned. Figure title and also the legend is mentioned. The legend identifies the visual elements used to distinguish different groups of data on the graph. The legend helps you evaluate the effects of grouping.

**Output:**



**Stacked Bar Graphs:**

A Stacked Bar graph is a graph that uses bars to show comparisons between categories of data, but with ability to break down and compare parts of a whole. Each bar in the chart represents a whole, and segments in the bar represent different parts or categories of that whole.

**Example:**

“import matplotlib.pyplot as plt

labels = ['G1', 'G2', 'G3', 'G4', 'G5']

men\_means = [20, 35, 30, 35, 27]

women\_means = [25, 32, 34, 20, 25]

width = 0.35 # the width of the bars: can also be len(x) sequence

fig, ax = plt.subplots()

ax.bar(labels, men\_means, width, label='Men')

ax.bar(labels, women\_means, width, bottom=men\_means, label='Women')

ax.set\_ylabel('Scores')

ax.set\_title('Scores by group and gender')

ax.legend()

y\_offset = -15

for bar in ax.patches:

ax.text(

*# Put the text in the middle of each bar. get\_x returns the start*

*# so we add half the width to get to the middle.*

bar.get\_x() + bar.get\_width() / 2,

*# Vertically, add the height of the bar to the start of the bar,*

*# along with the offset.*

bar.get\_height() + bar.get\_y() + y\_offset,

*# This is actual value we'll show.*

round(bar.get\_height()),

# Center the labels and style them a bit.

ha='center',

color='w',

weight='bold',

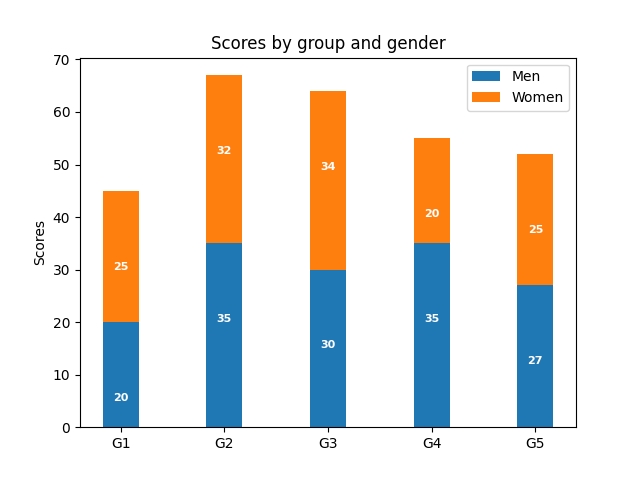
size=8

)

plt.show()”

The Stacked bar graphs are used to compare two or more categories or values and represent them in a single bar with different colors to identify each category. Here the men\_means and women\_means are the two lists that is going to be compared. The program is similar to that of the Grouped bar graph, including only one change in the “ax.bar” function. The 1st “ax.bar” represents the men\_means and the 2nd represents the women\_means where “bottom” is mentioned. Bottom is a keyword and also a parameter to the “ax.bar” function wherein the compared categories, which category should be in the bottom of the stack. As we mentioned “men\_means” in the bottom, in the bar the men\_means will be in the bottom and to its top women\_means. A looping statement “for” is used to mark the values of the corresponding bar and category inside its corresponding stack.

**Output:**



**Horizontal Bar Graph:**

In this type, the variables or the categories of the data have to be written and then the rectangular bars are horizontally drawn on the y-axis and the x-axis shows the length of the bars equal to the values of different variables present in the data.

**Example:**

“import matplotlib.pyplot as plt

import numpy as np

fig, ax = plt.subplots()

# Example data

people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')

y\_pos = np.arange(len(people))

performance = 3 + 10 \* np.random.rand(len(people))

ax.barh(y\_pos, performance, align='center')

ax.set\_yticks(y\_pos)

ax.set\_yticklabels(people)

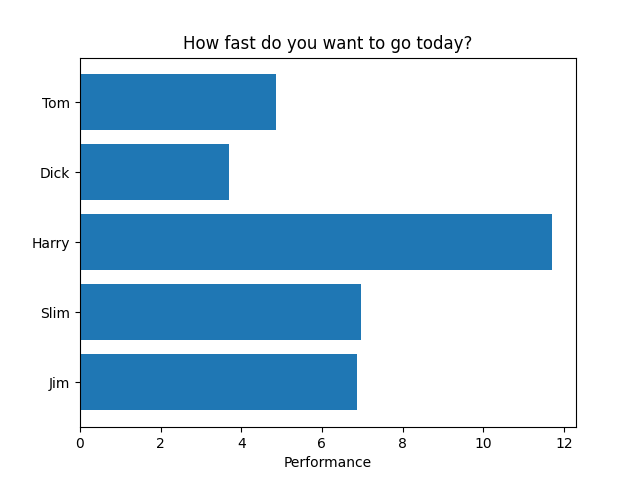
ax.invert\_yaxis() # labels read top-to-bottom

ax.set\_xlabel('Performance')

ax.set\_title('How fast do you want to go today?')

plt.show()”

The implementation of the horizontal bar graph is simple. The figure and axes are created. Then for y-axis a list of 5 people are considered and for x-axis, values are generated in a random manner and is stored in “performance” list. Usually, to represent bar graph “ax.bar()” is used. But in this case, “ax.barh()” is used, which plots the values in the y-axis. The bars are positioned at y with the given alignment. The horizontal baseline is left (default 0). Then plt.show() is used to represent the plotted graph.

**Output:**

**Scatter Plot:**

A scatter plot is a type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data. Scatter plots are primarily used  to observe and show relationships between two numeric variables. The dots in a scatter plot not only report the values of individual data points, but also patterns when the data are taken as a whole.

**Example:**

“import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]

y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)

plt.ylabel('Y')

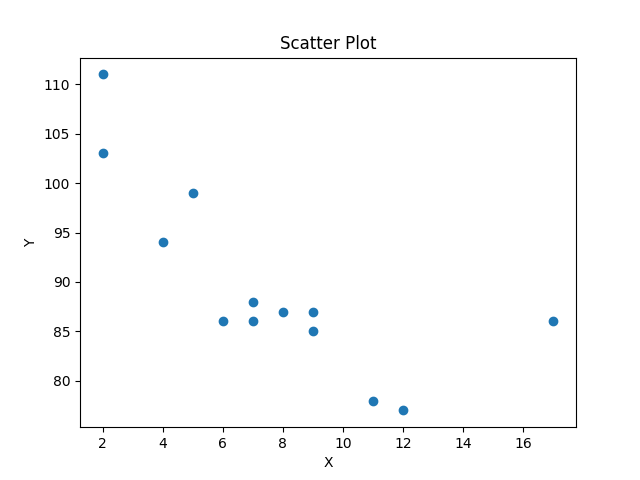
plt.xlabel("X")

plt.title("Scatter Plot")

plt.show()”

Two lists of equal length with some values are considered. It is plotted using “plt.scatter(list1,list2)” which plots the values of x and y in the corresponding coordinates. The label, title are given and the graph is shown using plt.show().

**Output:**



**Scatter Custom Symbol:**

Creating a scatter plot with custom symbol. There are many different types of symbols which can be used to plot in the graph. We are using a custom ellipse symbol to plot in the scatter plot.

**Example:**

“ import matplotlib.pyplot as plt

import numpy as np

# unit area ellipse

rx, ry = 3., 1.

area = rx \* ry \* np.pi

theta = np.arange(0, 2 \* np.pi + 0.01, 0.1)

verts = np.column\_stack([rx / area \* np.cos(theta), ry / area \* np.sin(theta)])

x, s = np.random.rand(2, 30)

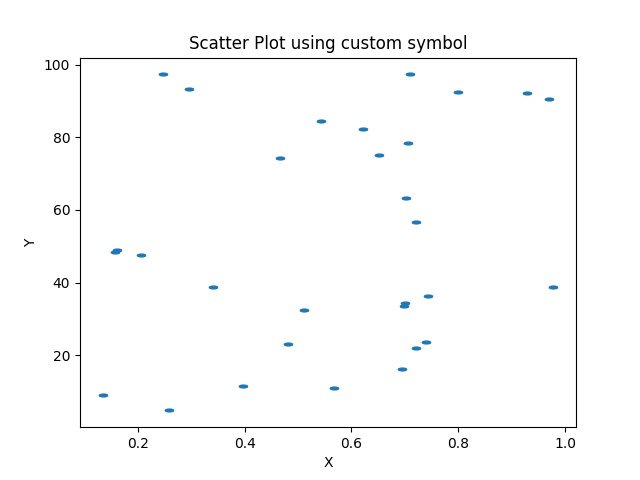
s \*= 10\*\*2.

fig, ax = plt.subplots()

ax.scatter(x, s, marker=verts)

plt.show()”

The area of the ellipse is fixed to mark in the graph. A variable verts is defined, where the measurements about the ellipse is stored which is then mentioned in the “ax.scatter()” as “marker=verts”. Then two lists x and s with random values are generated and is marked in the figure.

**Output:**

**Spectrum Representation:**

Matplotlib can also be used for spectrum representations. A signal is a function of time which can be represented by a series of sinusoidal components. These sinusoidal components have different amplitudes, different frequencies, different phases and magnitudes. Thus, the plotting of the frequency versus amplitude and the phase for the sinusoidal components comprise to form a signal called Frequency Spectrum or Spectrum of a signal. A Frequency Spectrum of a discrete-time signal is calculated by using the Fast Fourier Transform (FFT).

Fourier analysis of a periodic function refers to the extraction of the series of sines and cosines. This analysis can be expressed as Fourier Series. The Fast Fourier Transform (FFT) is a mathematical method for transforming a function of time into a function of frequency. It is very useful for time-dependent phenomena. The resulting spectrum shows the Frequency component of the given input signal.

**Example:**

“import matplotlib.pyplot as plt

import numpy as np

np.random.seed(0)

dt = 0.01 # sampling interval

Fs = 1 / dt # sampling frequency

t = np.arange(0, 10, dt)

# generate noise:

nse = np.random.randn(len(t))

r = np.exp(-t / 0.05)

cnse = np.convolve(nse, r) \* dt

cnse = cnse[:len(t)]

s = 0.1 \* np.sin(4 \* np.pi \* t) + cnse # the signal

fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(7, 7))

# plot time signal:

axs[0, 0].set\_title("Signal")

axs[0, 0].plot(t, s, color='C0')

axs[0, 0].set\_xlabel("Time")

axs[0, 0].set\_ylabel("Amplitude")

# plot different spectrum types:

axs[1, 0].set\_title("Magnitude Spectrum")

axs[1, 0].magnitude\_spectrum(s, Fs=Fs, color='C1')

axs[1, 1].set\_title("Log. Magnitude Spectrum")

axs[1, 1].magnitude\_spectrum(s, Fs=Fs, scale='dB', color='C1')

axs[2, 0].set\_title("Phase Spectrum ")

axs[2, 0].phase\_spectrum(s, Fs=Fs, color='C2')

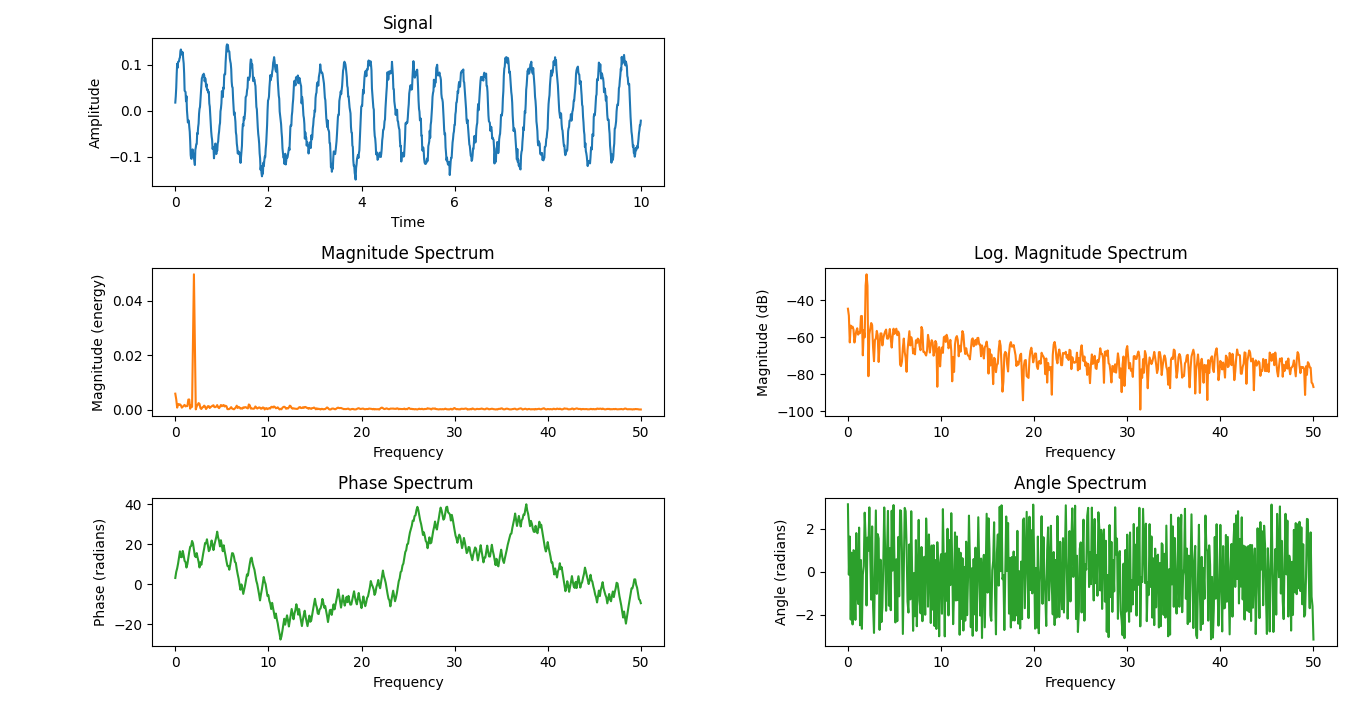
axs[2, 1].set\_title("Angle Spectrum")

axs[2, 1].angle\_spectrum(s, Fs=Fs, color='C2')

axs[0, 1].remove() # don't display empty ax

fig.tight\_layout()

plt.show()”

**Output:**

The plots shows the different spectrum graphs for a sine signal.

**Coherence of Two Signals:**

In signal processing, the coherence is the statistic value that can be used to examine the relationship between the two signals or data. Coherent signals are the signals that have same phase and frequency. The coherence function measures the correlation between two signals as a function of the frequency components which they contain. It is thus a correlation spectrum.

**Example:**

An example showing how to plot the coherence of two signals using python matplotlib.

“import numpy as np

import matplotlib.pyplot as plt

# Fixing random state for reproducibility

np.random.seed(19680801)

dt = 0.01

t = np.arange(0, 30, dt)

nse1 = np.random.randn(len(t)) # white noise 1

nse2 = np.random.randn(len(t)) # white noise 2

# Two signals with a coherent part at 10Hz and a random part

s1 = np.sin(2 \* np.pi \* 10 \* t) + nse1

s2 = np.sin(2 \* np.pi \* 10 \* t) + nse2

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

axs[0].plot(t, s1, t, s2)

axs[0].set\_xlim(0, 2)

axs[0].set\_xlabel('time')

axs[0].set\_ylabel('s1 and s2')

axs[0].set\_title('Two signals S1 and S2')

axs[0].grid(True)

cxy, f = axs[1].cohere(s1, s2, 256, 1. / dt)

axs[1].set\_ylabel('coherence')

fig.tight\_layout()

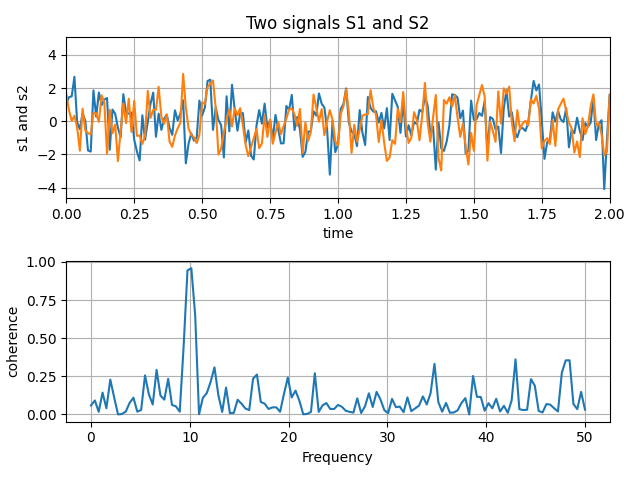
plt.show()”

Random values are obtained and formed as an array using numpy module. Two signals with a coherent part at 10Hz and a random part are created and is named as s1 and s2. Then an empty figure is created with 2 axes subplots. Then coherence between the two signals s1 and s2 are found using “axes.cohere” function in matplotlib.

The “axes.cohere()” function in axes module of matplotlib library is used to plot the coherence between two signals x and y.

Coherence is the normalizes cross spectral density: 13. Coherence formula.PNG

**Output:**



**Cross-Correlation Graph:**

Cross-Correlation is a measurement that tracks the movements of two or more sets of time series data relative to one another. It is used to compare multiple time series and determine how well they match up with each other and at what point the best match occurs. It is also known as sliding-dot product or sliding-inner product. The possible range of the correlation coefficient of the time series data is from -1 to +1. The closer the cross-correlation value is to 1, the more closely the sets are identical.

**Example:**

“import matplotlib.pyplot as plt

import numpy as np

# Fixing random state for reproducibility

np.random.seed(196)

x, y = np.random.randn(2, 100)

fig, ax1 = plt.subplots()

ax1.xcorr(x, y, usevlines=True, maxlags=50, normed=True, lw=2)

ax1.set\_title("Cross-Correlation")

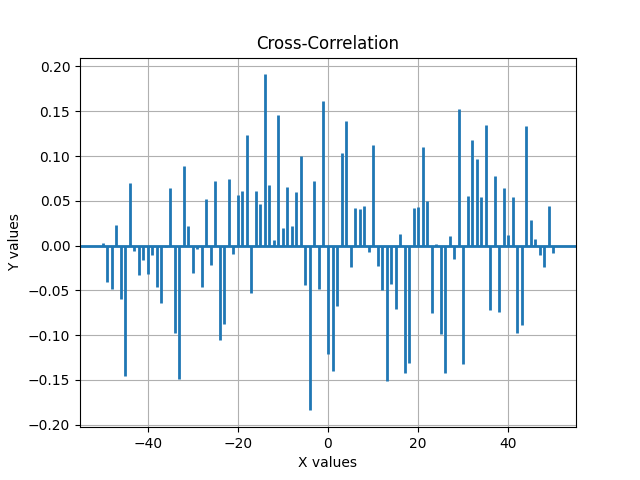
ax1.set\_xlabel("X values")

ax1.set\_ylabel("Y values")

ax1.grid(True)

plt.show()”

As usual, 2 sample lists with random values are created. The figure and an axes if formed. Then the values the two lists are plotted in a special manner using “ax1.xcorr()”. The “Axes.xcorr()” function is used to plot the correlation between x and y. Finally, the lables, titles are mentioned and it is shown using plt.show().

**Output:**

**Auto-Correleation Graph:**

Auto-Correlation is a mathematical representation of the degree of similarity between a given time series and a lagged version of itself over successive time intervals. Autocorrelation, sometimes known as serial correlation in the discrete time case, is the correlation of a signal with a delayed copy of itself as a function of delay.

**Example:**

“import matplotlib.pyplot as plt

import numpy as np

# Fixing random state for reproducibility

np.random.seed(1968)

x, y = np.random.randn(2, 100)

fig, ax1 = plt.subplots()

ax1.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)

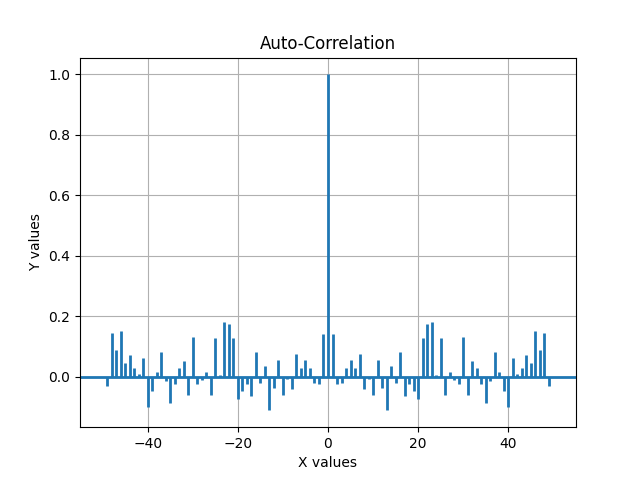
ax1.set\_title("Auto-Correlation")

ax1.set\_xlabel("X values")

ax1.set\_ylabel("Y values")

ax1.grid(True)

plt.show()”

**Output:**

**Changing Figure Size in Different Units:**

The figure size of a figure can be changed or can be set to a particular value using “plt.figure(figsize=(x,y))” function, where x, y are the size measurements given in inches which is default. The figure size can be given as the following units:

* Inches (default)
* Centimeters
* Pixels

Multiplying centimeter-based numbers with a conversion factor from cm to inches, gives the right numbers. Naming the conversion factor cm makes the conversion almost look like appending a unit to the number, which is nicely readable. Similarly, one can use a conversion from pixels.

Example:

“import matplotlib.pyplot as plt

text\_kwargs = dict(ha='center', va='center', fontsize=28, color='C1')

plt.subplots(figsize=(6, 2))

plt.text(0.5, 0.5, '6 inches x 2 inches', \*\*text\_kwargs)

cm = 1/2.54 # centimeters in inches

plt.subplots(figsize=(15\*cm, 5\*cm))

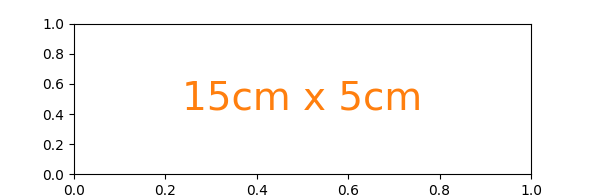
plt.text(0.5, 0.5, '15cm x 5cm', \*\*text\_kwargs)

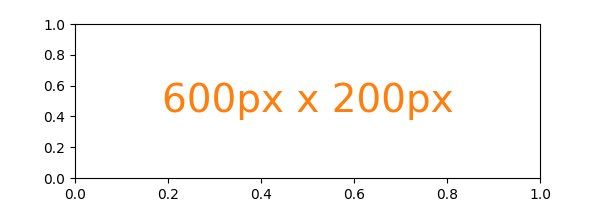
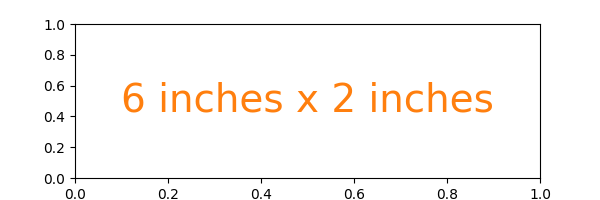
px = 1/plt.rcParams['figure.dpi'] # pixel in inches

plt.subplots(figsize=(600\*px, 200\*px))

plt.text(0.5, 0.5, '600px x 200px', \*\*text\_kwargs)

plt.show()”

**Output: **

****

**Different Scale:**

Two plots on the same axes with different left and right scales. The trick is to use two different axes that share the same x axis. You can use separate [matplotlib.ticker](https://matplotlib.org/stable/api/ticker_api.html" \l "module-matplotlib.ticker" \o "matplotlib.ticker) formatters and locators as desired since the two axes are independent. Such axes are generated by calling the [Axes.twinx](https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.twinx.html" \l "matplotlib.axes.Axes.twinx" \o "matplotlib.axes.Axes.twinx) method. Likewise, [Axes.twiny](https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.twiny.html" \l "matplotlib.axes.Axes.twiny" \o "matplotlib.axes.Axes.twiny) is available to generate axes that share a y axis but have different top and bottom scales.

**Example:**

import numpy as np

import matplotlib.pyplot as plt

# Create some mock data

t = np.arange(0.01, 10.0, 0.01)

data1 = np.exp(t)

data2 = np.sin(2 \* np.pi \* t)

fig, ax1 = plt.subplots()

plt.title("Plots with different scale")

color = 'tab:red'

ax1.set\_xlabel('time (s)')

ax1.set\_ylabel('exp', color=color)

ax1.plot(t, data1, color=color)

ax1.tick\_params(axis='y', labelcolor=color)

ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis

color = 'tab:blue'

ax2.set\_ylabel('sin', color=color) # we already handled the x-label with ax1

ax2.plot(t, data2, color=color)

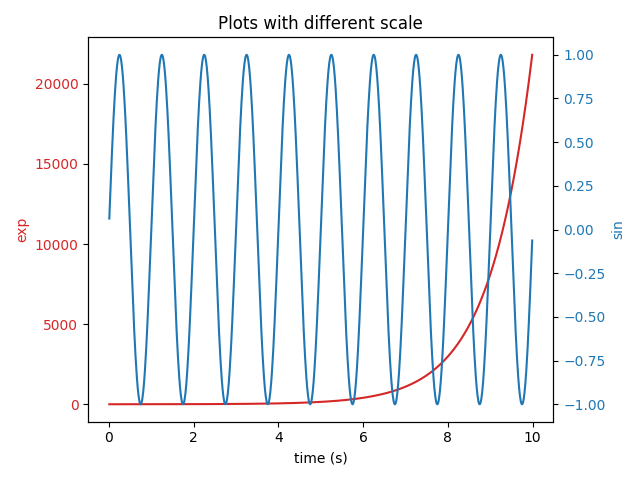
ax2.tick\_params(axis='y', labelcolor=color)

fig.tight\_layout() # otherwise the right y-label is slightly clipped

plt.show()

Two data lists are created using np.array. Then two subplots are created and the data lists are plotted in the same figure and identifying the 2 graphs by different colors. The tight layout is then shown as graph using plt.show().

**Output:**



**MRI with EEG in graph:**

In this method, the MRI image, its intensity and EEG can be plotted with the help of python’s matplotlib sub libraries. First the MRI image is read from the source using “matplotlib.cbook” module. Then the MRI image is loaded and is plotted on the figure. Then the given histogram of the MRI intensity is plotted. The EEG data is loaded from the source and is collected as a set of data. Then it is also plotted on the figure in a separate axes.

**Example:**

“import numpy as np

import matplotlib.pyplot as plt

import matplotlib.cbook as cbook

import matplotlib.cm as cm

from matplotlib.collections import LineCollection

from matplotlib.ticker import MultipleLocator

fig = plt.figure("MRI\_with\_EEG")

# Load the MRI data (256x256 16 bit integers)

with cbook.get\_sample\_data('s1045.ima.gz') as dfile:

im = np.frombuffer(dfile.read(), np.uint16).reshape((256, 256))

# Plot the MRI image

ax0 = fig.add\_subplot(2, 2, 1)

ax0.imshow(im, cmap=cm.gray)

ax0.axis('off')

# Plot the histogram of MRI intensity

ax1 = fig.add\_subplot(2, 2, 2)

im = np.ravel(im)

im = im[np.nonzero(im)] # Ignore the background

im = im / (2\*\*16 - 1) # Normalize

ax1.hist(im, bins=100)

ax1.xaxis.set\_major\_locator(MultipleLocator(0.4))

ax1.minorticks\_on()

ax1.set\_yticks([])

ax1.set\_xlabel('Intensity (a.u.)')

ax1.set\_ylabel('MRI density')

# Load the EEG data

n\_samples, n\_rows = 800, 4

with cbook.get\_sample\_data('eeg.dat') as eegfile:

data = np.fromfile(eegfile, dtype=float).reshape((n\_samples, n\_rows))

t = 10 \* np.arange(n\_samples) / n\_samples

# Plot the EEG

ticklocs = []

ax2 = fig.add\_subplot(2, 1, 2)

ax2.set\_xlim(0, 10)

ax2.set\_xticks(np.arange(10))

dmin = data.min()

dmax = data.max()

dr = (dmax - dmin) \* 0.7 # Crowd them a bit.

y0 = dmin

y1 = (n\_rows - 1) \* dr + dmax

ax2.set\_ylim(y0, y1)

segs = []

for i in range(n\_rows):

segs.append(np.column\_stack((t, data[:, i])))

ticklocs.append(i \* dr)

offsets = np.zeros((n\_rows, 2), dtype=float)

offsets[:, 1] = ticklocs

lines = LineCollection(segs, offsets=offsets, transOffset=None)

ax2.add\_collection(lines)

# Set the yticks to use axes coordinates on the y axis

ax2.set\_yticks(ticklocs)

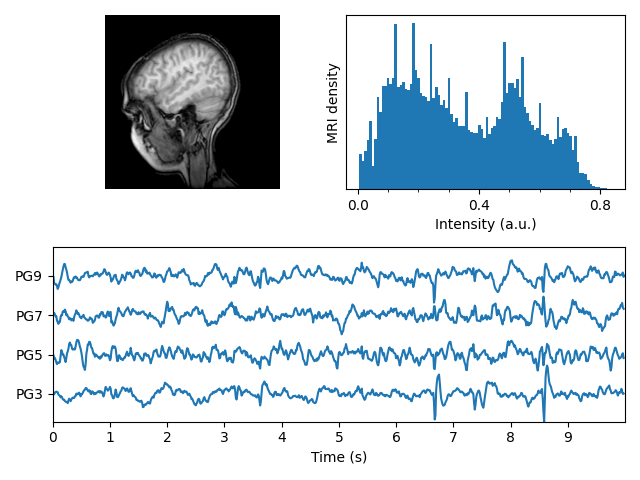
ax2.set\_yticklabels(['PG3', 'PG5', 'PG7', 'PG9'])

ax2.set\_xlabel('Time (s)')

plt.tight\_layout()

plt.show()”

**Output:**



**Pie Charts:**

A pie chart is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. In a pie chart, the arc length of each slice, is proportional to the quantity it represents .While it is named for its resemblance to a pie which has been sliced, there are variations on the way it can be presented.

Pie charts are widely used in the business world and mass media. They have been criticized and many experts recommend avoiding them as research has shown it is difficult to compare different sections of a given pie chart, or to compare data across different pie charts. Pie charts can be replaced in most cases by other plots such as the bar charts, box plots, dot plot, etc.

Matplotib API has a pie() function that generates a pie diagram representing data in an array. The fractional area of each wedge is given by x/sum(x).

If sum(x) < 1, then the values of x given the fractional area directly and the array will not be normalized.

**Program:**

“import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'

sizes = [15, 30, 45, 10]

explode = (0, 0.1, 0, 0) # only "explode" the 2nd slice (i.e. 'Hogs')

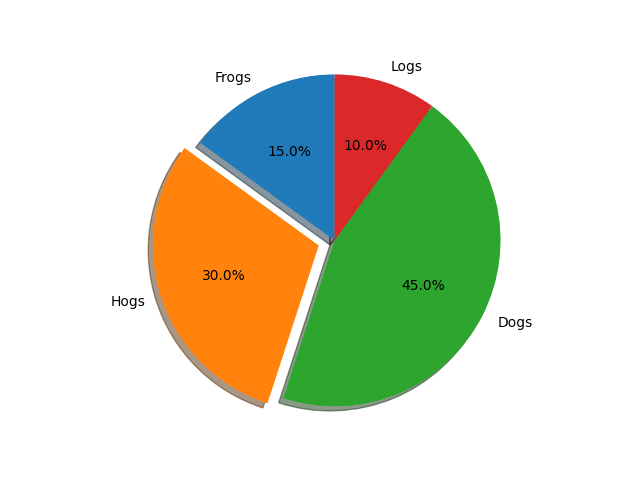
fig1, ax1 = plt.subplots()

ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startangle=90)

ax1.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()”

**Output:**



**Nested Pie charts:**

In this case, pie takes values corresponding to counts in a group. We'll first generate some fake data, corresponding to three groups. In the inner circle, we'll treat each number as belonging to its own group. In the outer circle, we'll plot them as members of their original 3 groups**.**

**Program:**

“import matplotlib.pyplot as plt

import numpy as np

fig, ax = plt.subplots()

size = 0.3

vals = np.array([[60., 32.], [37., 40.], [29., 10.]])

cmap = plt.get\_cmap("tab20c")

outer\_colors = cmap(np.arange(3)\*4)

inner\_colors = cmap([1, 2, 5, 6, 9, 10])

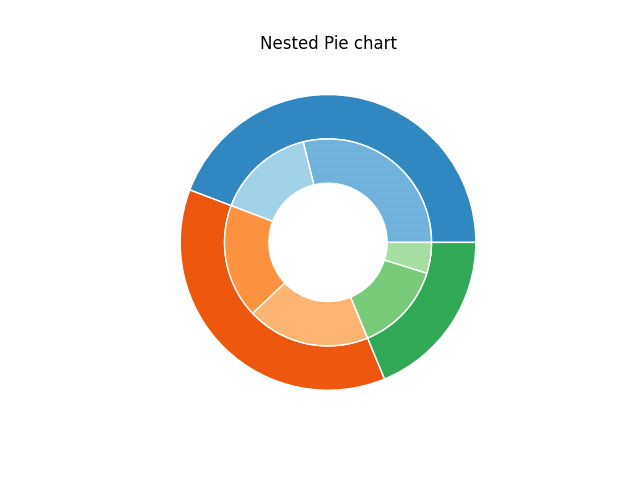
ax.pie(vals.sum(axis=1), radius=1, colors=outer\_colors, wedgeprops=dict(width=size, edgecolor='w'))

ax.pie(vals.flatten(), radius=1-size, colors=inner\_colors, wedgeprops=dict(width=size, edgecolor='w'))

ax.set(aspect="equal", title='Pie plot with `ax.pie`')

plt.show()”

**Output:**

****

**Labeling a pie and a donut:**

A donut chart is a variant of pie chart, with a hole in its center, and its displays categories as arcs rather than slices. Both make part-to-whole relationships easy to graphs at a glance.

As usual we would start by defining the imports and create a figure with subplots. Now it's time for the pie. Starting with a pie recipe, we create the data and a list of labels from it.

We can provide a function to the auto pct argument, which will expand automatic percentage labeling by showing absolute values; we calculate the latter back from relative data and the known sum of all values.

**Program for labeling a pie chart:**

“import numpy as np

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6, 3), subplot\_kw=dict(aspect="equal"))

recipe = ["375 g flour",

"75 g sugar",

"250 g butter",

"300 g berries"]

data = [float(x.split()[0]) for x in recipe]

ingredients = [x.split()[-1] for x in recipe]

def func(pct, allvals):

absolute = int(round(pct/100.\*np.sum(allvals)))

return "{:.1f}%\n({:d} g)".format(pct, absolute)

wedges, texts, autotexts = ax.pie(data, autopct=lambda pct: func(pct, data),

textprops=dict(color="w"))

ax.legend(wedges, ingredients,

title="Ingredients",

loc="center left",

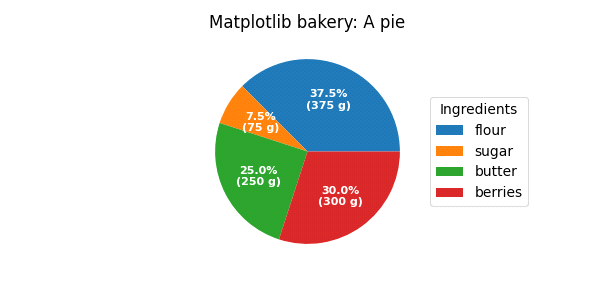
bbox\_to\_anchor=(1, 0, 0.5, 1))

plt.setp(autotexts, size=8, weight="bold")

ax.set\_title("Matplotlib bakery: A pie")

plt.show()”

**Output:**

****

**Program for labelling a donut chart:**

“import numpy as np

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6, 3), subplot\_kw=dict(aspect="equal"))

recipe = ["225 g flour",

"90 g sugar",

"1 egg",

"60 g butter",

"100 ml milk",

"1/2 package of yeast"]

data = [225, 90, 50, 60, 100, 5]

wedges, texts = ax.pie(data, wedgeprops=dict(width=0.5), startangle=-40)

bbox\_props = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)

kw = dict(arrowprops=dict(arrowstyle="-"),

bbox=bbox\_props, zorder=0, va="center")

for i, p in enumerate(wedges):

ang = (p.theta2 - p.theta1)/2. + p.theta1

y = np.sin(np.deg2rad(ang))

x = np.cos(np.deg2rad(ang))

horizontalalignment = {-1: "right", 1: "left"}[int(np.sign(x))]

connectionstyle = "angle,angleA=0,angleB={}".format(ang)

kw["arrowprops"].update({"connectionstyle": connectionstyle})

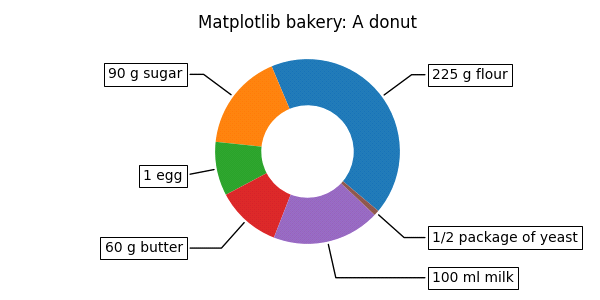
ax.annotate(recipe[i], xy=(x, y), xytext=(1.35\*np.sign(x), 1.4\*y),

horizontalalignment=horizontalalignment, \*\*kw)

ax.set\_title("Matplotlib bakery: A donut")

plt.show()”

**Output:**

****

**Style sheets:**

A web style sheet is a form of separation of presentation and content for web design in which the markup of a webpage contains the page's semantic content and structure, but does not define its visual layout. There are many types of style sheets and two of those are explained from the style sheets.

**Five Thirty Eight Style Sheet:**

FiveThirtyEight, sometimes rendered as 538, is an American website that **focuses on opinion poll analysis, politics, economics, and sports blogging**.

**Program:**

“import matplotlib.pyplot as plt

import numpy as np

plt.style.use('fivethirtyeight')

x = np.linspace(0, 10)

np.random.seed(19680801)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))

ax.plot(x, np.sin(x) + 0.5 \* x + np.random.randn(50))

ax.plot(x, np.sin(x) + 2 \* x + np.random.randn(50))

ax.plot(x, np.sin(x) - 0.5 \* x + np.random.randn(50))

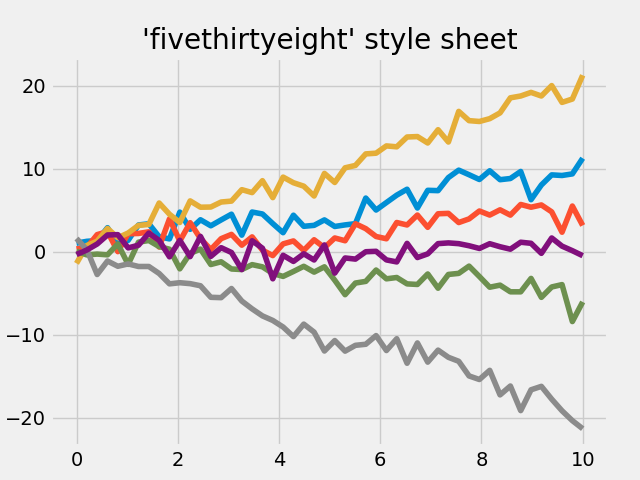
ax.plot(x, np.sin(x) - 2 \* x + np.random.randn(50))

ax.plot(x, np.sin(x) + np.random.randn(50))

ax.set\_title("'fivethirtyeight' style sheet")

plt.show()”

**Output:**

****

**Solarized light Style Sheet:**

It’s another kind of graph that is used for plotting values with random lines.

**Program:**

“import matplotlib.pyplot as plt

import numpy as np

np.random.seed(19680801)

x = np.linspace(0, 10)

with plt.style.context('Solarize\_Light2'):

plt.plot(x, np.sin(x) + x + np.random.randn(50))

plt.plot(x, np.sin(x) + 2 \* x + np.random.randn(50))

plt.plot(x, np.sin(x) + 3 \* x + np.random.randn(50))

plt.plot(x, np.sin(x) + 4 + np.random.randn(50))

plt.plot(x, np.sin(x) + 5 \* x + np.random.randn(50))

plt.plot(x, np.sin(x) + 6 \* x + np.random.randn(50))

plt.plot(x, np.sin(x) + 7 \* x + np.random.randn(50))

plt.plot(x, np.sin(x) + 8 \* x + np.random.randn(50))

# Number of accent colors in the color scheme

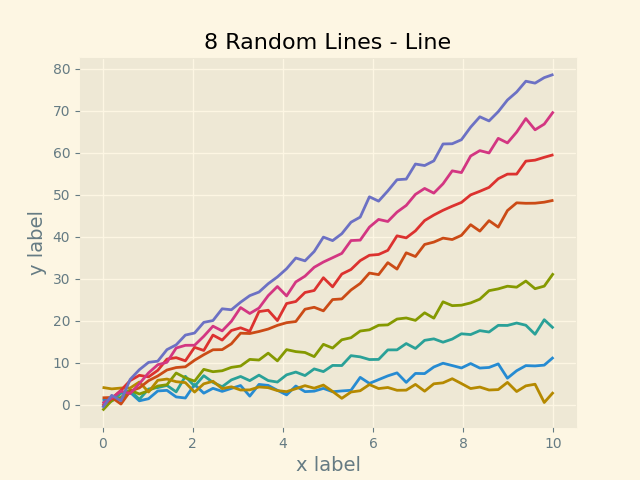
plt.title('8 Random Lines - Line')

plt.xlabel('x label', fontsize=14)

plt.ylabel('y label', fontsize=14)

plt.show()”

**Output:**

****

**3D Graphs:**

**Three-dimensional space** is a geometric setting in which three values are required to determine the position of an element. This is the informal meaning of the term [dimension](https://en.wikipedia.org/wiki/Dimension). In geometry, a three-dimensional shape can be defined as a solid figure or an object or shape that has three dimensions – length, width and height. In 3D graphs we are about to see some graphs here.

**Plot 2D data on 3D plot:**

**Program:**

“import numpy as np

import matplotlib.pyplot as plt

ax = plt.figure().add\_subplot(projection='3d')

# Plot a sin curve using the x and y axes.

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

y = np.sin(x \* 2 \* np.pi) / 2 + 0.5

ax.plot(x, y, zs=0, zdir='z', label='curve in (x, y)')

# Plot scatterplot data (20 2D points per colour) on the x and z axes.

colors = ('r', 'g', 'b', 'k')

# Fixing random state for reproducibility

np.random.seed(19680801)

x = np.random.sample(20 \* len(colors))

y = np.random.sample(20 \* len(colors))

c\_list = []

for c in colors:

c\_list.extend([c] \* 20)

# By using zdir='y', the y value of these points is fixed to the zs value 0

# and the (x, y) points are plotted on the x and z axes.

ax.scatter(x, y, zs=0, zdir='y', c=c\_list, label='points in (x, z)')

# Make legend, set axes limits and labels

ax.legend()

ax.set\_xlim(0, 1)

ax.set\_ylim(0, 1)

ax.set\_zlim(0, 1)

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

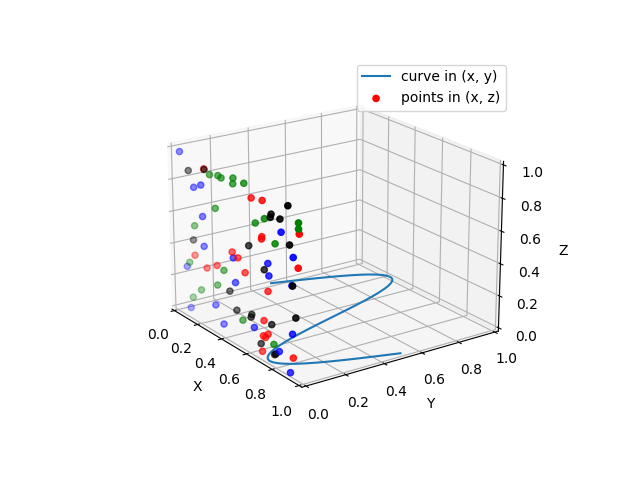
# Customize the view angle so it's easier to see that the scatter points lie

# on the plane y=0

ax.view\_init(elev=20., azim=-35)

plt.show()”

**Output:**

****

**Create 2D bar graphs in different planes:**

**Program:**

“import matplotlib.pyplot as plt

import numpy as np

# Fixing random state for reproducibility

np.random.seed(19680801)

fig = plt.figure()

ax = fig.add\_subplot(projection='3d')

colors = ['r', 'g', 'b', 'y']

yticks = [3, 2, 1, 0]

for c, k in zip(colors, yticks):

# Generate the random data for the y=k 'layer'.

xs = np.arange(20)

ys = np.random.rand(20)

# You can provide either a single color or an array with the same length as

# xs and ys. To demonstrate this, we color the first bar of each set cyan.

cs = [c] \* len(xs)

cs[0] = 'c'

# Plot the bar graph given by xs and ys on the plane y=k with 80% opacity.

ax.bar(xs, ys, zs=k, zdir='y', color=cs, alpha=0.8)

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

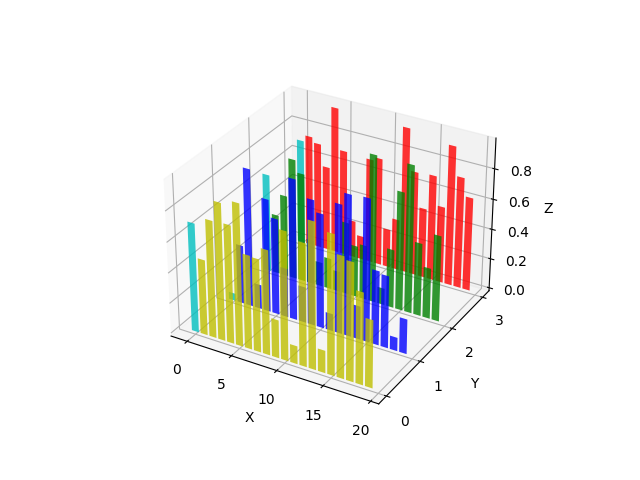
ax.set\_zlabel('Z')

# On the y axis let's only label the discrete values that we have data for.

ax.set\_yticks(yticks)

plt.show()”

**Output:**

****

**Create 3D histogram of 2D data:**

**Program:**

“import matplotlib.pyplot as plt

import numpy as np

# Fixing random state for reproducibility

np.random.seed(19680801)

fig = plt.figure()

ax = fig.add\_subplot(projection='3d')

x, y = np.random.rand(2, 100) \* 4

hist, xedges, yedges = np.histogram2d(x, y, bins=4, range=[[0, 4], [0, 4]])

# Construct arrays for the anchor positions of the 16 bars.

xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing="ij")

xpos = xpos.ravel()

ypos = ypos.ravel()

zpos = 0

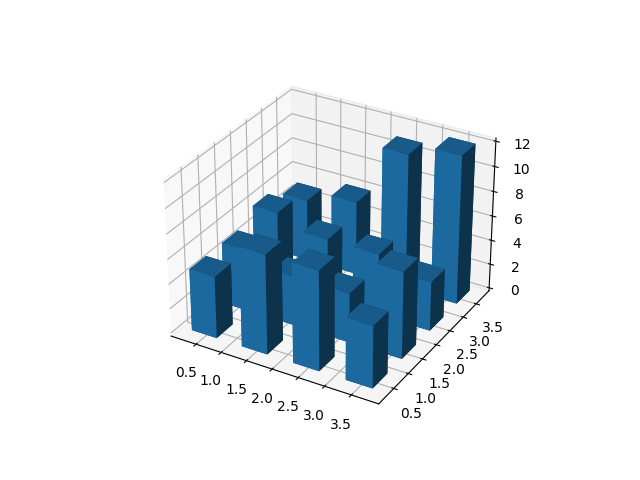
# Construct arrays with the dimensions for the 16 bars.

dx = dy = 0.5 \* np.ones\_like(zpos)

dz = hist.ravel()

ax.bar3d(xpos, ypos, zpos, dx, dy, dz, zsort='average')

plt.show()”

**Output:**

**3D surface:**

# Program:

# “import matplotlib.pyplot as plt

# from matplotlib import cm

# from matplotlib.ticker import LinearLocator

# import numpy as np

# fig, ax = plt.subplots(subplot\_kw={"projection": "3d"})

# # Make data.

# X = np.arange(-5, 5, 0.25)

# Y = np.arange(-5, 5, 0.25)

# X, Y = np.meshgrid(X, Y)

# R = np.sqrt(X\*\*2 + Y\*\*2)

# Z = np.sin(R)

# # Plot the surface.

# surf = ax.plot\_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False)

# # Customize the z axis.

# ax.set\_zlim(-1.01, 1.01)

# ax.zaxis.set\_major\_locator(LinearLocator(10))

# # A StrMethodFormatter is used automatically

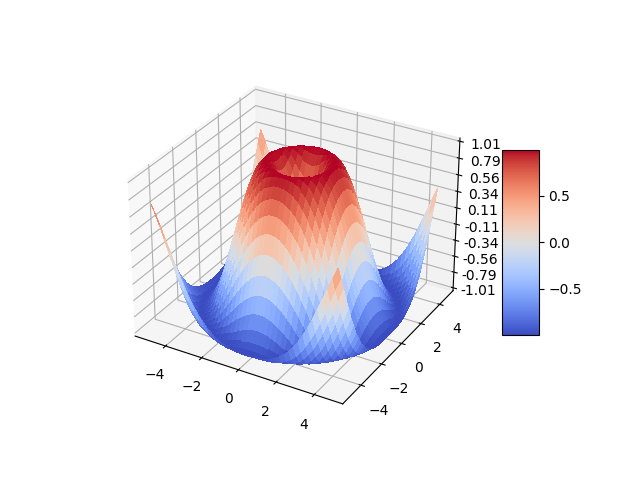
# ax.zaxis.set\_major\_formatter('{x:.02f}')

# # Add a color bar which maps values to colors.

# fig.colorbar(surf, shrink=0.5, aspect=5)

# plt.show()”

**Output:**

****

**Animation:**

Matplotlib has a very special function, animation. The easiest way to make a live animation in matplotlib is to use the Animation classes.

**Live Line Graph:**

In this Matplotlib Animation function, an example of how to create live updating graphs that can update their plots live as the data-source updates. You may want to use this for something like graphing live stock pricing data, or maybe you have a sensor connected to your computer, and you want to display the live sensor data. To do this, we use the animation functionality with Matplotlib.

**Example:**

“import matplotlib.pyplot as plt

import matplotlib.animation as animation

from matplotlib import style

style.use('fivethirtyeight')

fig = plt.figure()

ax1 = fig.add\_subplot(1,1,1)

def animate(i):

graph\_data = open('example.txt','r').read()

lines = graph\_data.split('\n')

xs = []

ys = []

for line in lines:

if len(line) > 1:

x, y = line.split(',')

xs.append(float(x))

ys.append(float(y))

ax1.clear()

ax1.plot(xs, ys)

ani = animation.FuncAnimation(fig, animate, interval=1000)

plt.show()”

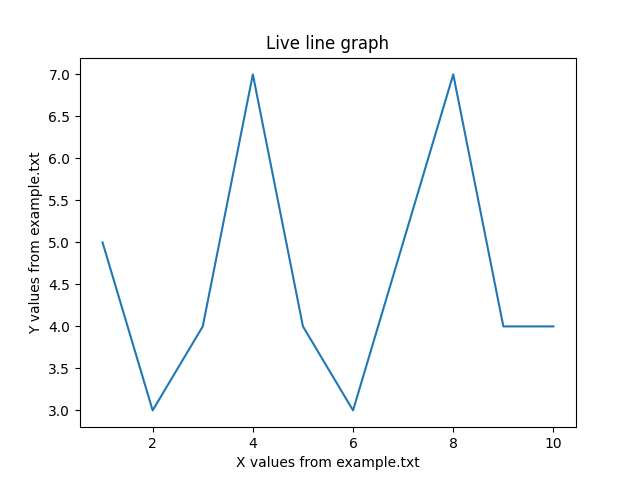
Here, the only new import is the matplotlib.animation as animation. This is the module that will allow us to animate the figure. In animate function, what we’re doing is building the data and then plotting it in the figure. A notepad file named ‘example.txt’ contains the following data :

[(1,5),(2,3),(3,4),(4,7),(5,4),(6,3),(7,5),(8,7),(9,4),(10,4)]

This data stored in the notepad is updated in realtime and the graph plots its point accordingly.

This is the function of the animate function from Matplotlib.

**Output:**



The values get updated from the ‘example.txt’ file and the graph moves on according to the x-axis.

**Oscilloscope Live:**

Oscilloscopes test and display voltage signals as waveforms, visual representations of the variation of voltage over time. The signals are plotted on a graph, which shows how the signal changes. The vertical (Y) access represents the voltage measurement and the horizontal (X) axis represents time. One of the biggest advantages to using oscilloscopes is the fact that they give real-time analysis. This means that you get the right reading once you connect the device to a power source. This oscilloscope graph can be implemented using matplotlib.

**Implementation:**

“import numpy as np

from matplotlib.lines import Line2D

import matplotlib.pyplot as plt

import matplotlib.animation as animation

from IPython import display

class Scope:

def \_\_init\_\_(self, ax, maxt=2, dt=0.02):

self.ax = ax

self.dt = dt

self.maxt = maxt

self.tdata = [0]

self.ydata = [0]

self.line = Line2D(self.tdata, self.ydata)

self.ax.add\_line(self.line)

self.ax.set\_ylim(-.1, 1.1)

self.ax.set\_xlim(0, self.maxt)

self.ax.set\_xlabel("Time in secs")

self.ax.set\_ylabel("Voltage")

self.ax.set\_title("Live Oscilloscope")

def update(self, y):

lastt = self.tdata[-1]

if lastt > self.tdata[0] + self.maxt: # reset the arrays

self.tdata = [self.tdata[-1]]

self.ydata = [self.ydata[-1]]

self.ax.set\_xlim(self.tdata[0], self.tdata[0] + self.maxt)

self.ax.figure.canvas.draw()

t = self.tdata[-1] + self.dt

self.tdata.append(t)

self.ydata.append(y)

self.line.set\_data(self.tdata, self.ydata)

return self.line,

def emitter(p=0.1):

"""Return a random value in [0, 1) with probability p, else 0."""

while True:

v = np.random.rand(1)

if v > p:

yield 0.

else:

yield np.random.rand(1)

# Fixing random state for reproducibility

np.random.seed(19680801 // 10)

fig, ax = plt.subplots()

scope = Scope(ax)

# pass a generator in "emitter" to produce data for the update func

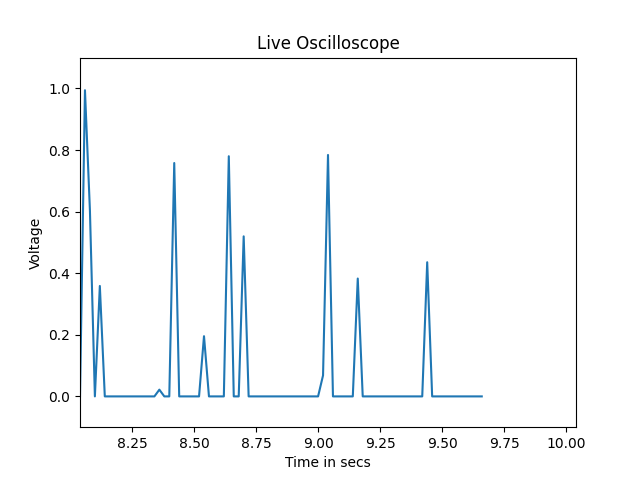
ani = animation.FuncAnimation(fig, scope.update, emitter, interval=50, blit=True)

plt.show()”

The entire program is divided into two parts. The main and the class Scope. The class Scope has few sub functions defined within it. There are 2 functions inside the class scope namely \_\_init\_\_, update. The \_\_init\_\_ function acts as the main function inside the class scope. The subplot is created and the x,y limits, labels are defined inside the \_\_init\_\_ function.

The Update function plays a major role in the animation. It gets the values generated by the emitter function which acts as a generator and generates random values which is given to the update function. The update function plots the graphs accordingly with the incoming values generated by the Emitter function.

**Output:**



The values gets updated in the plot and since the x axis limit is set to particular value, the axis gets updated too accordingly.

Thus, with the help of Matplotlib Library from Python, many types of graphs have been created and explained. Matplotlib is a python plotting package that makes it simple to create two-dimensional and three-dimensional plots from data stored in a variety of data structures including lists, numpy arrays and pandas dataframes. “matplotlib.pyplot” is a collection of 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, plots some lines in a plotting area, decorates the plot with labels, etc.

**Matplotlib Keywords:**

There are various keywords under matplotlib python, each carrying a specific function.

**Matplotlib.pyplot:**

**Pyplot function overview:**

Matplotlib.pypllot is a state-based interface to matplotlib.

Source : “https://matplotlib.org/stable/api/pyplot\_summary.html”

“Matplotlib.pyplot.plotting()”

|  |  |
| --- | --- |
| **Funtion** | **Description** |
| *acorr* | Plot the autocorrelation of x |
| arrow | Add arrow to the axes |
| autoscale | Autoscale the axis to view the data (toggles the axis) |
| axes | Add an axes to the current figure and makes it current axes |
| axis | Convenience method to get or set some axes properties |
| bar | Make a bar chart |
| bar\_label | Give a label to the bar chart |
| barh | Make a horizontal bar graph |
| cla | Clear the current axis |
| clim | Set the color limit to the current image |
| close | Close the figure window |
| cohere | Plot the coherence between x and y |
| colorbar | Add a colorbar to the plot |
| contour | Plot the contour lines |
| contourf | Plot the filled contour |
| draw | Redraw the current figure |
| draw\_if\_interactive | Redraw the current figure if in interactive mode |
| figimage | Add a non-resampled image to the figure |
| figlegend | Place a legend on the figure |
| fignum\_exists | Return if the figure with given id exists |
| figtext | Add a text to the figure |
| figure | Create a new figure or activate an existing figure |
| gca | Get current axes, creating one if necessary |
| gcf | Get current figure |
| gci | Get current colorable artist |
| get\_figlabels | Return a list of existing figure labels |
| get\_fignums | Return a list of existing figure numbers |
| grid | Configure grid lines |
| hist | Plot a histogram |
| hist2d | Make a 2D histogram plot |
| imread | Read an image from from a file into an array |
| imsave | Save an array as an image file |
| imshow | Display data as an image (2d regular) |
| ioff | Disable interactive mode |
| ion | Enable interactive mode |
| legend | Place legend on the axes |
| pie | Plot a pie chart |
| plot | Plot y versus x as lines or markers |
| polar | Make a polar plot |
| savefig | Save the current figure |
| scatter | A scatter plot of y versus x |
| show | Display all open figures |
| subplot | Add an axes to the current figure or retrieve an existing figure |
| subplots | Create a figure and a number of subplots |
| suptitle | Add a centered suptitle to the figure |
| table | Add a table to the Axes |
| text | Add a text to the Axes |
| tick\_params | Change the appearance of ticks, tick labels and gridlines |
| tight\_layout | Adjusting the padding between and around subplots |
| title | Set the title to the Axes |
| twinx | Make and return a second axes that shares the x-axis |
| twiny | Make and return a second axes that shares the y-axis |
| xcorr | Plot the corss correlation between x and y |
| xlabel | Set the label for the x-axis |
| xlim | Get or Set the x limits for the current axes |
| xscale | Set the x-axis scale |
| xticks | Get or set the current tick locations and labels for y-axis |
| ylabel | Set the label for the y-axis |
| ylim | Get or Set the y limits for the current axes |
| yscale | Set the y-axis scale |
| yticks | Get or set the current tick locations and labels for y-axis |