**Visualization with Matplotlib and Seaborn**

**5.1 General Matplotlib Tips**

**Basic definitions**

* **Figure –** It is a whole figure which contains one or more than one axes or plots.
* **Axes –** A figure contains usually more than one axes (plots) and it may contain two or three in case of three-dimensional structure or objects. Each Axes has a title, an X –label, and Y –label.
* **Axis-** It takes care of generating graph limits
* **Artists-** Mostly they are tied to the axes, and whatever we see on the figure like text objects, line2d objects, collection objects.

**How to display your plots**

**The following are the ways to display the plots in 3 different contexts**

**1.Plotting from a Script**

Plt.show() can be used to visualize the plot

**Example -**

Import matplotlib.pyplot as plt

Import numpy as np

x=np.linespace(0,10,100)

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

plt.show()

**2.Plotting from a IPython Shell**

**We need to specify the command %matplotlib% and then import the library**

**Example –**

%matplotlib%

Import matplotlib.pyplot as plt

Any Plt plot command will cause a window to open which will contain the figure.Plt.draw() needs to be used to force an update to the figure already drawn to reflect the changes .

**3.Plotting from a IPython notebook**

We can use the commands ‘%matplotlib notebook’ which will lead to have interactive plots within the notebook. We can use ‘%matplotlib inline’to view static images of the plot.

Example

%matplotlib inline

Import numpy as np

x=np.linespace(0,10,100)

fig = plt.figure()

plt.plot(x,np(sin(x),’-’)

## Stateful Versus Stateless Approaches

## Stateful services keep track of sessions or transactions and react differently to the same inputs based on that history,There is only ever one Figure or Axes that you’re manipulating at a given time, and you don’t need to explicitly refer to it. Stateless services rely on clients to maintain sessions and center around operations that manipulate resources, rather than the state, Modifying the underlying objects directly is the object-oriented approach. We usually do this by calling methods of an Axes object, which is the object that represents a plot itself.

## 5.2 Simple Line Plots

We need to first import the library then we can use plt.title to give a title to the plot, xand y label to provide a label to the x and y axis. show() function can be used to display the plot.

The color property can be used to specify different colors.

Ex - plt.plot([10,20,9,16],[23,45,56,100], color = ‘blue’)

The Linestyle property can be used to specify different linestyles such as ‘Solid’,’dashed’ etc.

Ex - plt.plot([10,20,9,16],[23,45,56,100], linestyle = ‘dashed’)

**Example –**

Import matplotlib.pyplot as plt

Import numpy as np

plt.plot([10,20,9,16],[23,45,56,100] , color=’blue’,linestyle=’solid’)

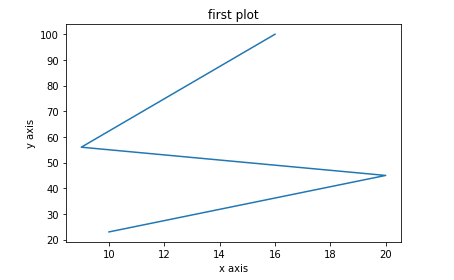
plt.title(“first plot”)

plt.xlabel(“x axis”)

plt.ylabel(“y axis”)

plt.show()

So, here we are passing two arrays as input and using show(), to get the required plot, as from the graph you can see that the first array appears on the x-axis and the second array appears on the y-axis of the given plot.



**Adjusting the Plot : Axes Limits**

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

**Example –**

Import matplotlib.pyplot as plt

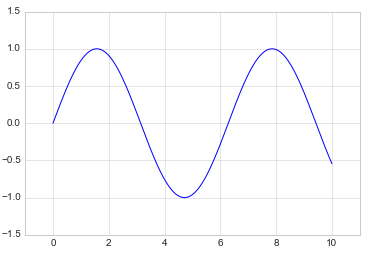
Import numpy as np

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

plt.xlim(-1, 11)

plt.ylim(-1.5, 1.5)

plt.show()



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

Ex - plt.axis([-1, 11, -1.5, 1.5])

**Labelling Plots**

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

Ex - plt.plot(x, np.sin(x))

plt.title("A Sine Curve")

plt.xlabel("x")

plt.ylabel("sin(x)")

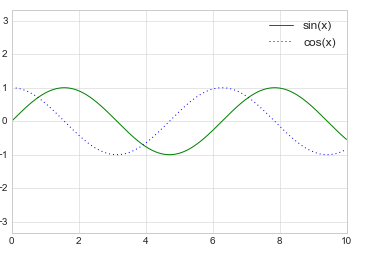
When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. Again, Matplotlib has a built-in way of quickly creating such a legend. It is done via theplt.legend() method

Ex - plt.plot(x, np.sin(x), '-g', label='sin(x)')

plt.plot(x, np.cos(x), ':b', label='cos(x)')

plt.axis('equal')

plt.legend();



**5.3Simple Scatter Plots**

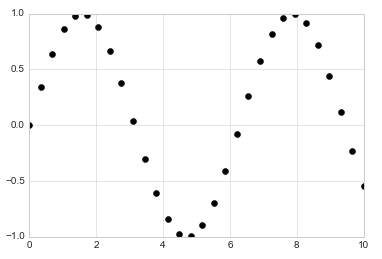
Another commonly used plot type is the simple scatter plot, here the points are represented individually with a dot, circle, or other shape. The plt.plot/ax.plot  can produce scatter plots as well.

Example

x = np.linspace(0, 10, 30)

y = np.sin(x)

plt.plot(x, y, 'o', color='black');



The third argument in the function call is a character that represents the type of symbol used for the plotting, different symbols can be used here.

A variety of options are available to customize the plots such as color,markersize,markeredgewidth etc.

Example

plt.plot(x, y, '-p', color='gray',

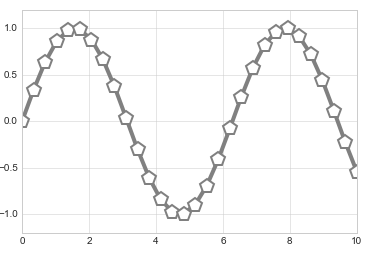
markersize=15, linewidth=4,

markerfacecolor='white',

markeredgecolor='gray',

markeredgewidth=2)

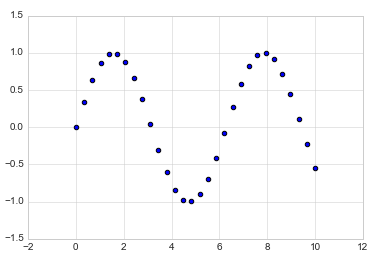
plt.ylim(-1.2, 1.2);



**Scatter Plots with plt.scatter**

plt.scatter function, which can be used very similarly to the plt.plot function.

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



The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

To demonstrate the flexibility of plt.scatterfunction by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the alpha keyword to adjust the transparency level.

Example -

rng = np.random.RandomState(0)

x = rng.randn(100)

y = rng.randn(100)

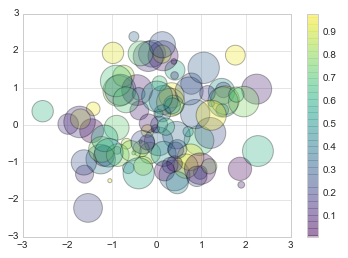
colors = rng.rand(100)

sizes = 1000 \* rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,

cmap='viridis')

plt.colorbar(); # show color scale



plt.plot can be noticeably more efficient than plt.scatter. The reason is that plt.scatter has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In plt.plot, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data.

# **5.4Visualizing Errors**

To visualize the error that’s possible in a calculation can be displayed by an error bar. A basic errorbar can be created with a single Matplotlib function call:

Example

%matplotlib inline

import matplotlib.pyplot as plt

plt.style.use('seaborn-whitegrid')

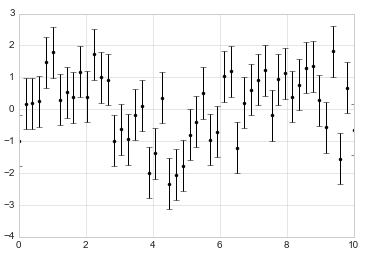
import numpy as np

x = np.linspace(0, 10, 50)

dy = 0.8

y = np.sin(x) + dy \* np.random.randn(50)

plt.errorbar(x, y, yerr=dy, fmt='.k')



In addition to these basic options, the errorbar function has many options to fine-tune the outputs. Using these additional options you can easily customize the aesthetics of your errorbar plot.

Ex - plt.errorbar(x, y, yerr=dy, fmt='o', color='black', ecolor='lightgray', elinewidth=3, apsize=0)

## Continuous Errors

In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like plt.plot and plt.fill\_between for a useful result.

Example

Here we'll perform a simple Gaussian process regression, using the Scikit-Learn API. This is a method of fitting a very flexible non-parametric function to data with a continuous measure of the uncertainty

from sklearn.gaussian\_process import GaussianProcess

# define the model and draw some data

model = lambda x: x \* np.sin(x)

xdata = np.array([1, 3, 5, 6, 8])

ydata = model(xdata)

# Compute the Gaussian process fit

gp = GaussianProcess(corr='cubic', theta0=1e-2, thetaL=1e-4, thetaU=1E-1,

random\_start=100)

gp.fit(xdata[:, np.newaxis], ydata)

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

yfit, MSE = gp.predict(xfit[:, np.newaxis], eval\_MSE=True)

dyfit = 2 \* np.sqrt(MSE) # 2\*sigma ~ 95% confidence region

We now have xfit, yfit, and dyfit, which sample the continuous fit to our data. We could pass these to the plt.errorbar function as above, but we don't really want to plot 1,000 points with 1,000 errorbars. Instead, we can use the plt.fill\_between function with a light color to visualize this continuous error

# Visualize the result

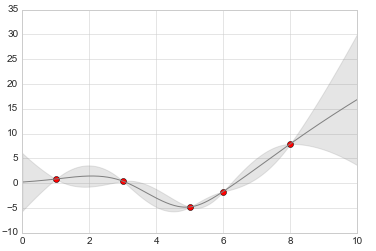
plt.plot(xdata, ydata, 'or')

plt.plot(xfit, yfit, '-', color='gray')

plt.fill\_between(xfit, yfit - dyfit, yfit + dyfit,

color='gray', alpha=0.2)

plt.xlim(0, 10);



**5.5 Density and Contour Plots**

Contour plots (sometimes called Level Plots) are a way to show a three-dimensional surface on a two-dimensional plane. There are three Matplotlib functions that can be helpful for this task: plt.contour for contour plots, plt.contourf for filled contour plots, and plt.imshow for showing images

A contour plot can be created with the plt.contour function. It takes three arguments: a grid of *x* values, a grid of *y* values, and a grid of *z* values. The *x* and *y* values represent positions on the plot, and the *z* values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays

**Example**

%matplotlib inline

import matplotlib.pyplot as plt

plt.style.use('seaborn-white')

import numpy as np

def f(x, y):

return np.sin(x) \*\* 10 + np.cos(10 + y \* x) \* np.cos(x)

x = np.linspace(0, 5, 50)

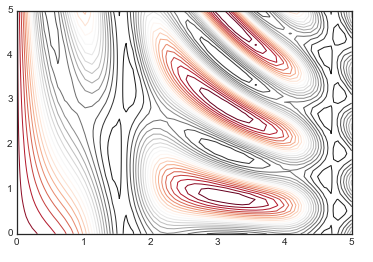
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)

Z = f(X, Y)

plt.contour(X, Y, Z, colors='black');

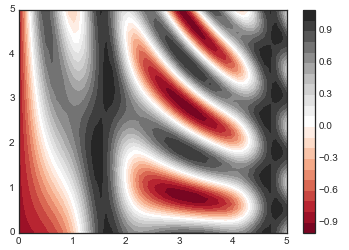
plt.contour(X, Y, Z, 20, cmap='RdGy');



We can switch to a filled contour plot using the plt.contourf() function (notice the f at the end), which uses largely the same syntax as plt.contour().Additionally, we'll add a plt.colorbar() command, which automatically creates an additional axis with labeled color information for the plot.

plt.contourf(X, Y, Z, 20, cmap='RdGy')

plt.colorbar()



The colorbar makes it clear that the black regions are "peaks," while the red regions are "valleys."One potential issue with this plot is that , the color steps are discrete rather than continuous, which is not always what is desired. This could be remedied by setting the number of contours to a very high number, but this results in a rather inefficient plot: Matplotlib must render a new polygon for each step in the level. A better way to handle this is to use the plt.imshow() function, which interprets a two-dimensional grid of data as an image

Example

plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',

cmap='RdGy')

plt.colorbar()

plt.axis(aspect='image');

* plt.imshow() doesn't accept an x and y grid, so you must manually specify the extent [xmin, xmax, ymin, ymax] of the image on the plot.
* plt.imshow() by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.
* plt.imshow() will automatically adjust the axis aspect ratio to match the input data; this can be changed by setting, for example, plt.axis(aspect='image') to make x and y units match.