Module 5
Visualization with Matplotlib and Seaborn

Data Visualization: Matplotlib package – Plotting Graphs – Controlling Graph – Adding Text – More Graph Types – Getting and setting values – Patches. Advanced data visualization with Seaborn.- Time series analysis with Pandas.

Matplotlib package

Matplotlib is a multiplatform data visualization library built on NumPy arrays. The matplotlib package is the main graphing and plotting tool . The package is versatile and highly configurable, supporting several graphing interfaces.

Matplotlib, together with NumPy and SciPy provides MATLAB-like graphing capabilities.

The benefits of using matplotlib in the context of data analysis and visualization are as follows:

- Plotting data is simple and intuitive.
- Performance is great; output is professional.
- Integration with NumPy and SciPy (used for signal processing and numerical analysis) is seamless.
- The package is highly customizable and configurable, catering to most people's needs.

The package is quite extensive and allows embedding plots in a graphical user interface.

Other Advantages

- One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib.
- It has led to a *large userbase*, which in turn has led to an active developer base and Matplotlib's powerful tools and ubiquity within the scientific Python world.
- Pandas library itself can be used as wrappers around Matplotlib's API. Even with wrappers like these, it is still often useful to dive into Matplotlib's syntax to adjust the final plot output.

Plotting Graphs

- Line plots, scatter plots, bar charts, histograms, etc.

This section details the building blocks of plotting graphs: the plot() function and how to control it to generate the output we require.

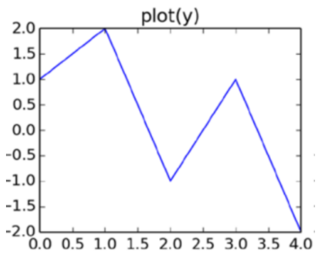
The plot() function is highly customizable, accommodating various options, including plotting lines and/or markers, line widths, marker types and sizes, colors, and a legend to associate with each plot.

The functionality of plot() is similar to that of MATLAB and GNU-Octave with some minor differences, mostly due to the fact that Python has a different syntax from MATLAB and GNU-Octave.

Lines and Markers

- Lets begin by creating a vector to plot using NumPy

```
>>> figure()
>>> y = array([1, 2, -1, 1])
>>> plot(y)
>>> show()
```

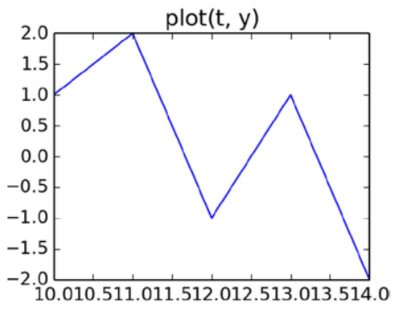


The vector y is passed as an input to plot(). As a result, plot() drew a graph of the vector y using auto-incrementing integers for an x-axis. Which is to say that, if x-axis values are not supplied, plot() will automatically generate one for you: plot(y) is equivalent to plot(range(len(y)), y).

Note:If you don't have a GUI installed with matplotlib, replace show() with savefig('filename') and open the generated image file in an image viewer.)

- let's supply x-axis values (denoted by variable t):

```
>>> figure()
>>> y = array([1, 2, -1, 1])
>>> t = array([10, 11, 12, 13])
>>> plot(t, y)
>>> show()
```



The call to function figure() generates a new figure to plot on, so we don't overwrite the previous figure.

- Let's look at some more options. Next, we want to plot y as a function of t, but display only markers, not lines. This is easily done:

```
>>> figure()
>>> plot(t, y, 'o')
>>> show()
```

To select a different marker, replace the character 'o' with another marker symbol.

Table below lists some popular choices; issuing help(plot) provides a full account of the available markers.

Table 6-2. *A Sampling of Available Plot Markers*

Character	Marker Symbol
'o'	Circle
'^'	Upward-pointing triangle
's'	Square
'+'	Plus
'x'	Cross (multiplication)
'D'	Diamond

Controlling Graph

- Axis limits, labels, ticks, colors, styles, etc.
- Use keyword arguments when calling plotting functions in Matplotlib.

For a graph to convey an idea aesthetically, though it is important, the data is not everything. The grid and grid lines, combined with a proper selection of axis and labels, present additional layers of information that add clarity and contribute to overall graph presentation.

Now, let's focus to controlling the figure by controlling the x-axis and y-axis behavior and setting grid lines.

- Axis
- Grid and Ticks
- Subplots
- Erasing the Graph

Axis

The axis() function controls the behavior of the x-axis and y-axis ranges. If you do not supply a parameter to axis(), the return value is a tuple in the form (xmin, xmax, ymin, ymax). You can use axis() to set the new axis ranges by specifying new values: axis([xmin, xmax, ymin, ymax]).

If you'd like to set or retrieve only the x-axis values or y-axis values, do so by using the functions xlim(xmin, xmax) or ylim(ymin, ymax), respectively.

The function axis() also accepts the following values: 'auto', 'equal', 'tight', 'scaled', and 'off'.

- The value '**auto**'—the default behavior—allows plot() to select what it thinks are the best values.
- The value '**equa**l' forces each x value to be the same length as each y value, which is important if you're trying to convey physical distances, such as in a GPS plot.
- The value '**tight**' causes the axis to change so that the maximum and minimum values of x and y both touch the edges of the graph.
- The value '**scaled**' changes the x-axis and y-axis ranges so that x and y have both the same length (i.e., aspect ratio of 1).
- Lastly, calling **axis('off')** removes the axis and labels.

To illustrate these axis behaviors, a circle is plotted as below:

Listing 6-3. Plotting a Circle

```
R = 1.2
I = arange(0, 2*pi, 0.01)
plot(sin(I)*R, cos(I)*R)
show()
```

Figure below shows the results of applying different axis values to this circle.

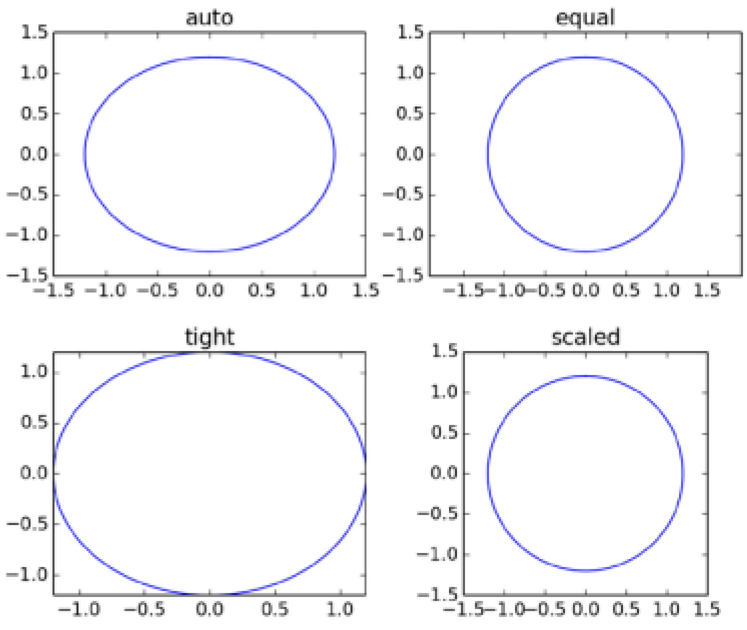


Figure 6-4. *Controlling axis behavior*

Grid and Ticks

The function grid() draws a grid in the current figure. The grid is composed of a set of horizontal and vertical dashed lines coinciding with the x ticks and y ticks. You can toggle the grid by calling grid() or set it to be either visible or hidden by using grid(True) or grid(False), respectively.

To control the ticks (and effectively change the grid lines, as well), use the functions xticks() and yticks(). The functions behave similarly to axis() in that they return the current ticks if no parameters are passed; you can also use these functions to set ticks once parameters are provided. The functions take an array holding the tick values as numbers and an optional tuple containing text labels. If the tuple of labels is not provided, the tick numbers are used as labels.

Listing 6-4. Grid and Tick Example

```
R = 1.2
I = arange(0, 4*pi, 0.01)
plot(sin(I)*R, cos(0.5*I)*R)
axhline(color='gray')
axvline(color='gray')
grid()
xticks([-1, 0, 1], ('Negative', 'Neutral', 'Positive'))
yticks(arange(-1.5, 2.0, 1))
show()
```

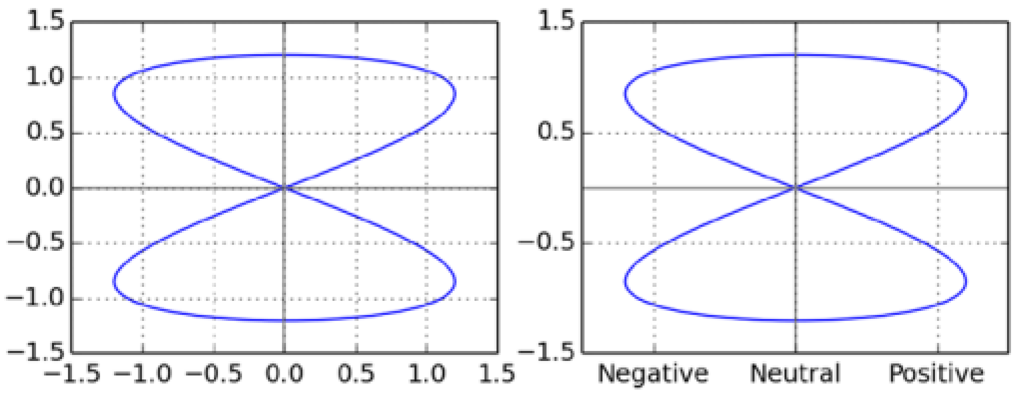


Figure 6-5. *Controlling the grid and axis: the left graph shows the default `xticks()`, while the right graph displays labels*

Adding Text

There are several options to annotate your graph with text. You've already seen some, such as using the xticks() and yticks() function.

The following functions will give you more control over text in a graph.

Title

The function title(str) sets str as a title for the graph and appears above the plot area. The function accepts the arguments listed in Table 6-5.

Table 6-5. *Text Arguments*

Argument	Description	Values
`fontsize`	Controls the font size	`'large'`, `'medium'`, `'small'`, or an actual size (i.e., 50)
`verticalalignment` or `va`	Controls the vertical alignment	`'top'`, `'baseline'`, `'bottom'`, `'center'`
`horizontalalignment` or `ha`	Controls the horizontal alignment	`'center'`, `'left'`, `'right'`

All alignments are based on the default location, which is centered above the graph. Thus, setting ha='left' will print the title starting at the middle (horizontally) and extending to the right. Similarly, setting ha='right' will print the title ending in the middle of the graph (horizontally). The same applies for vertical alignment. Here's an example of using the title() function:

```
>>> title('Left aligned, large title', fontsize=24, va='baseline')
```

Axis Labels and Legend

The functions xlabel() and ylabel() are similar to title(), only they're used to set the x-axis and y-axis labels, respectively. Both these functions accept the text arguments .

```
>>> xlabel('time [seconds]')
```

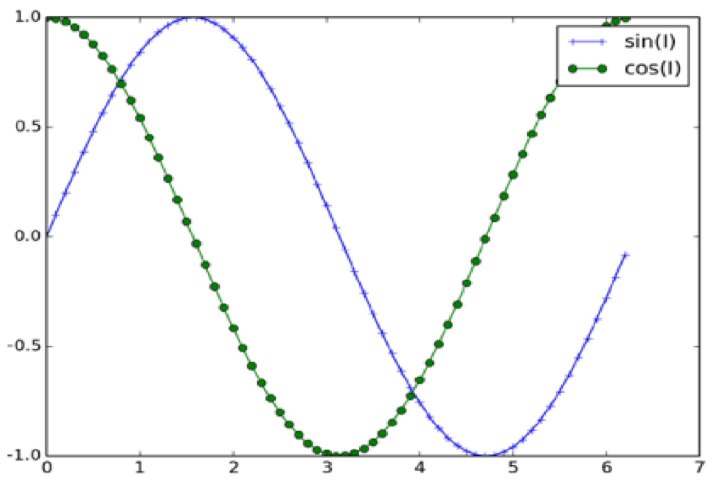
Next on our list of text functions is legend(). The legend() function adds a legend box and associates a plot with text:

```
>>> I = arange(0, 2*pi, 0.1)
>>> plot(I, sin(I), '+-', I, cos(I), 'o-')
>>> legend(['sin(I)', 'cos(I)'])
>>> show()
```

The legend order associates the text with the plot. An alternative approach is to specify the label argument with the plot() function call, and then issue a call to legend() with no parameters:

```
>>> I = arange(0, 2*pi, 0.1)
>>> plot(I, sin(I), '+-', label='sin(I)')
>>> plot(I, cos(I), 'o-', label='cos(I)')
>>> legend()
>>> show()
```

Figure 6-7 shows the addition of an x-axis label and legend.



loc can take one of the following values: 'best', 'upper right', 'upper left', 'lower left', 'lower right', 'right', 'center left', 'center right', 'lower center', 'upper center', and 'center'. Instead of using strings, use numbers: 'best' corresponds to 0, 'upper left' corresponds to 1, and 'center' corresponds to 10. Using the value 'best' moves the legend to a spot less likely to hide data; however, performance-wise there may be some impact.

Text Rendering

The text(x, y, str) function accepts the coordinates in graph units x, y and the string to print, str. It also renders the string on the figure. You can modify the text alignment using the arguments. The following will print text at location (0, 0):

```
>>> figure()
>>> plot([-1, 1], [-1, 1])
>>> text(0, 0, 'origin', va='center', ha='center')
>>> show()
```

The function text() has many other arguments, such as rotation and fontsize.

Example:

The example script summarizes the functions we've discussed up to this point: plot() for plotting; title(), xlabel(), ylabel(), and text() for text annotations; and xticks(), ylim(), and grid() for grid control.

Listing 6-6. A Plot Summary Example

```
I = arange(0, 2*pi+0.1, 0.1)
plot(I, sin(I), label='sin(I)')
title('Function f(x)=sin(x)')
xlabel('x [rad]', va='bottom')
ylabel('sin(x)')
text(pi/2, 1, 'Max value', ha='center', va='bottom')
text(3*pi/2, -1, 'Min value', ha='center', va='top')
xticks(linspace(pi/2, 2*pi, 4), (r'$\frac{\pi}{2}$', r'$\pi$', \
    r'$\frac{3\pi}{2}$', r'$2 \pi$'), fontsize=20)
xlim([0, 2*pi])
ylim([-1.2, 1.2])
grid()
show()
```

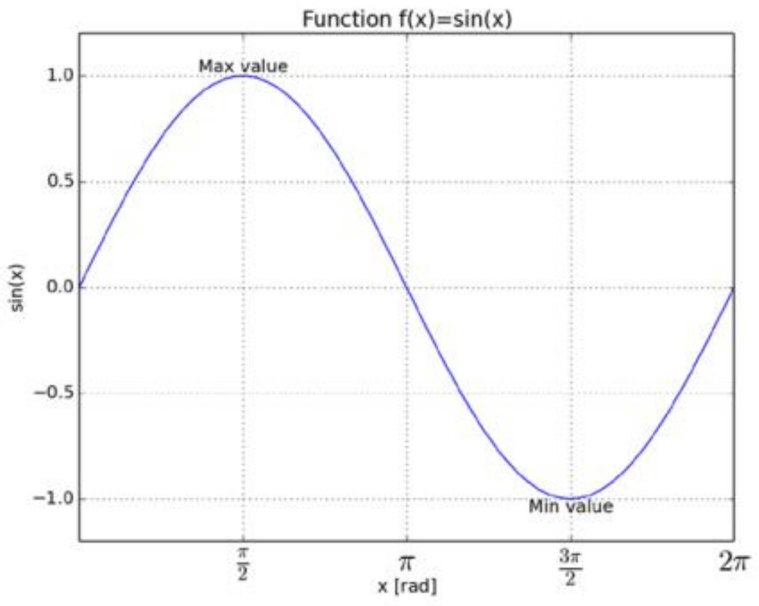


***Figure 6-8.** A plot summary example*

More Graph Types

- Boxplots, pie charts, polar plots, 3D plots, etc.
- Matplotlib offers a variety of specialized plotting functions for different types of data.

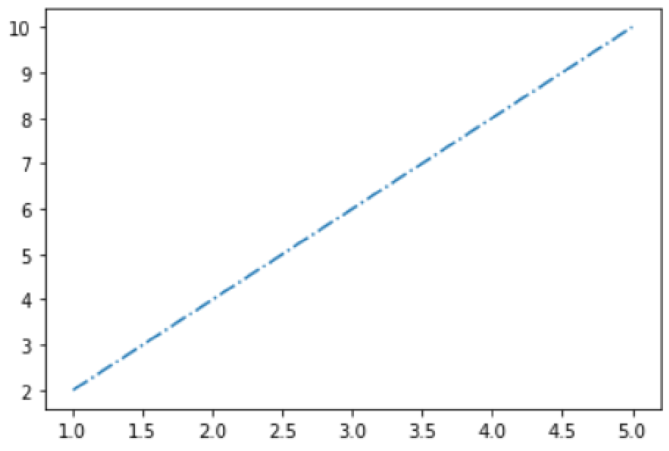
(Note: Refer to the text book)

Getting and setting values

Object-oriented design of matplotlib involves two functions, setp() and getp(), that retrieve and set a matplotlib object's parameters. The benefit of using setp() and getp() is that automation is easily achieved. Whenever a plot() command is called, matplotlib returns a list of matplotlib objects.

For example, you can use the getp() function to get the linestyle of a line object. You can use the setp() function to set the linestyle of a line object.

Here is an example of how to use the getp() and setp() functions to get and set the linestyle of a line object:

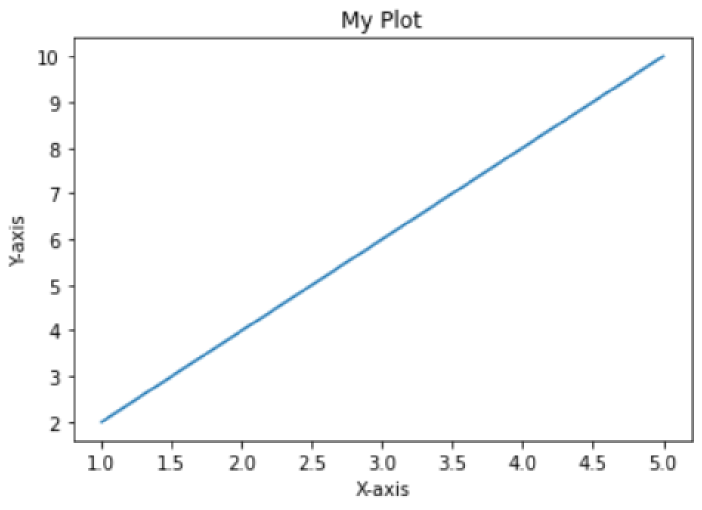
```
import matplotlib.pyplot as plt
# Create a figure and a set of subplots
fig, ax = plt.subplots()

# Generate some data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Plot the data
line, = ax.plot(x, y)

# Get the linestyle of the line object
linestyle = plt.getp(line, 'linestyle')

# Print the linestyle
print('Linestyle:', linestyle)

# Set the linestyle to dashed
plt.setp(line, linestyle=' -.')

# Show the plot
plt.show()
```
	Linestyle: - 

This code will create a line plot of the data in the x and y lists. The linestyle of the line object will be set to dashed. The code will then print the linestyle to the console. Finally, the code will show the plot.

Here is a Python code example of getting and setting values in Matplotlib:

```python
import matplotlib.pyplot as plt

# Create a figure and a set of subplots
fig, ax = plt.subplots()

# Generate some data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Plot the data
ax.plot(x, y)

# Set the x-axis label
ax.set_xlabel('X-axis')

# Set the y-axis label
ax.set_ylabel('Y-axis')

# Set the title of the plot
ax.set_title('My Plot')

# Get the x-axis limits
x_min, x_max = ax.get_xlim()

# Get the y-axis limits
y_min, y_max = ax.get_ylim()

# Print the x-axis limits
print('X-axis limits:', x_min, x_max)

# Print the y-axis limits
print('Y-axis limits:', y_min, y_max)

# Show the plot
plt.show( )
```

```
X-axis limits: 0.8 5.2
Y-axis limits: 1.6 10.4
```



This code will create a line plot of the data in the x and y lists. The x-axis label will be set to "X-axis", the y-axis label will be set to "Y-axis", and the title of the plot will be set to "My Plot". The code will then print the x-axis and y-axis limits to the console. Finally, the code will show the plot.

You can use the get_xlim() and get_ylim() functions to get the current x-axis and y-axis limits, respectively. You can use the set_xlim() and set_ylim() functions to set the x-axis and y-axis limits, respectively.

Patches

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as ***patches***. Some of these, like Rectangle and Circle are found in matplotlib.pyplot, but the full set is located in matplotlib.patches.

To use patches, follow these steps:

1. Draw a graph.
2. Create a patch object.
3. Attach the patch object to the figure using the add_patch() function.

To add a shape to a plot, create the patch object *shp* and add it to a subplot by calling ax.add_patch(*shp*).

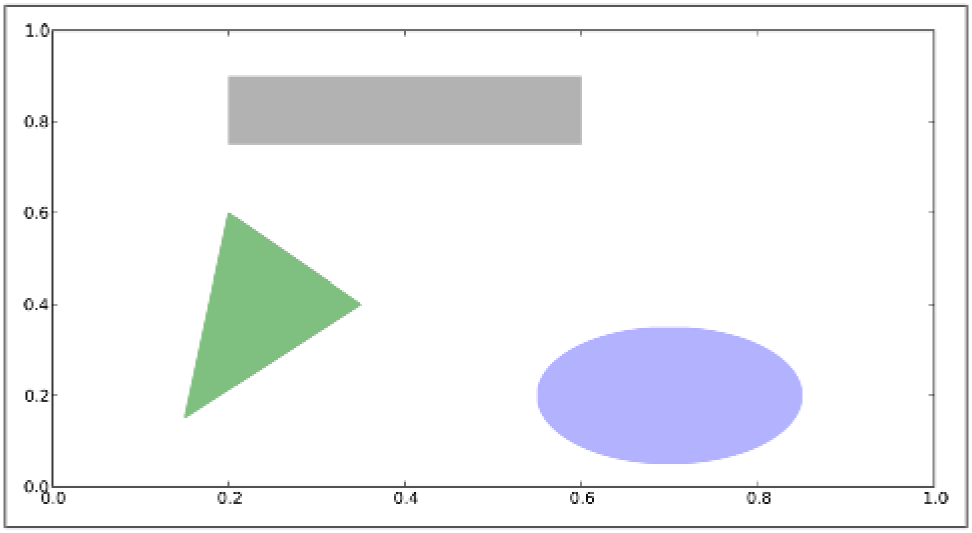


Figure 8-12. Figure composed from 3 different patches

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]], color='g', alpha=0.5)
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

To work with patches, assign them to an already existing graph because, in a sense, patches are “patched” on top of a figure. Table below gives a partial listing of available patches. In this table, the notation xy indicates a list or tuple of (x, y) values

Table 6-6. *Available Patches*

Patch	Description
`Arrow(x, y, dx, dy)`	An arrow, starting at location (x, y) and ending at location (x+dx, y+dy)
`Circle(xy, r)`	A circle centered at xy and radius r
`Ellipse(xy, w, h, angle)`	An ellipse centered at xy, of width w, height h, and rotated angle degrees
`Polygon([xy1, xy2, xy3,...])`	A polygon made of vertices specified by xy points
`Wedge(xy, r, theta1, theta2)`	A wedge (part of a circle) centered at xy, of radius r, starting at angle theta1 and ending at angle theta2
`Rectangle(xy, w, h)`	A rectangle, starting at xy, of width w and height h

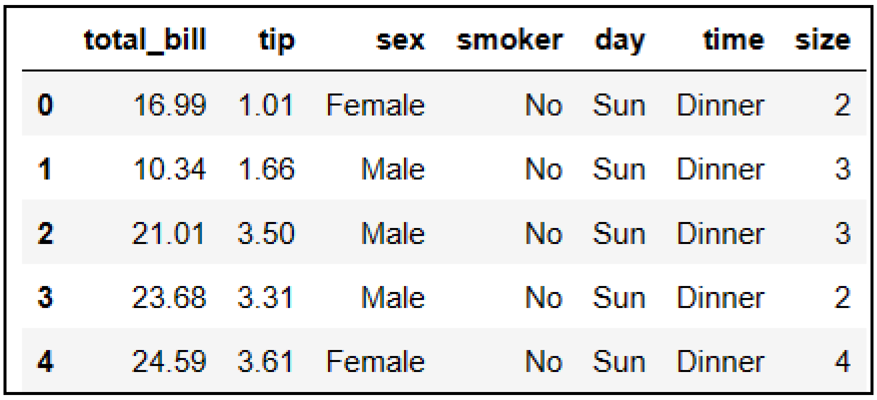
Advanced data visualization with Seaborn

Seaborn is a powerful Python data visualization library based on Matplotlib. It provides a high-level interface for creating attractive and informative statistical graphics. Here's a guide for advanced data visualization with Seaborn:

Import Libraries: Import the necessary libraries including Seaborn and Matplotlib. Seaborn comes with several built-in datasets for practice. You can load one using the **load_dataset** function.

```
import seaborn as sns
import matplotlib.pyplot as plt
tips_data = sns.load_dataset("tips")
tips_data.head()
```

Data set has the following columns :



	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Load a Dataset:

```
tips_data = sns.load_dataset("tips")
```

Customize Seaborn Styles: Seaborn comes with several built-in styles. You can set the style using sns.set_style().

```
sns.set_style("whitegrid")
# Other styles include "darkgrid", "white", "dark", and "ticks"
```

Advanced Scatter Plots: Create a scatter plot with additional features like hue, size, and style.

```
sns.scatterplot(x="total_bill", y="tip", hue="day", size="size", style="sex", data=tips_data)
```

Pair Plots for Multivariate Analysis: Visualize relationships between multiple variables with pair plots.

```
sns.pairplot(tips_data, hue="sex", markers=["o", "s"], palette="husl")
```

Heatmaps: Create a heatmap to visualize the correlation matrix of variables.

```
correlation_matrix = tips_data.corr()
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm")
```

Violin Plots: Visualize the distribution of a numerical variable for different categories.

```
sns.violinplot(x="day", y="total_bill", hue="sex", data=tips_data, split=True, inner="quart")
```

FacetGrid for Customized Subplots: Use **FacetGrid** to create custom subplots based on categorical variables.

```
g = sns.FacetGrid(tips_data, col="time", row="smoker", margin_titles=True)
g.map(sns.scatterplot, "total_bill", "tip")
```

Joint Plots: Combine scatter plots with histograms for bivariate analysis.

```
sns.jointplot(x="total_bill", y="tip", data=tips_data, kind="hex")
```

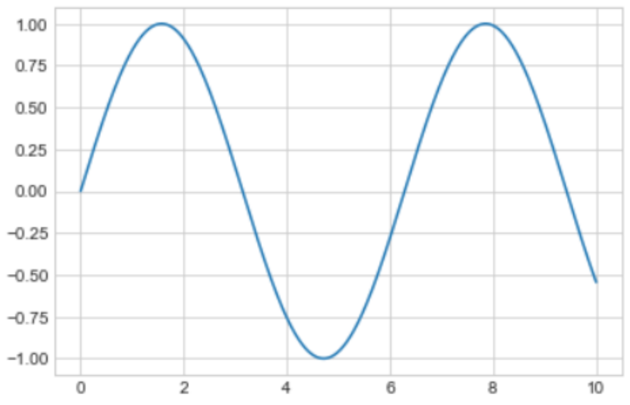
Question bank:

1. What are patches? Explain with an example.
2. How to get and set the values in the graphs? Give example
3. Discuss any 3 aspects of the graph that can be controlled to enhance the visualization.
4. How to annotate the graph with text. Illustrate with example.
5. What is Seaborn? List the advantages.
6. Write a Python code to plot following graphs using Seaborn:
   a) line plot b) histogram c) scatter plot
7. What is Time series analysis? Write a Python program to demonstrate Timeseries analysis with Pandas.
8. Discuss Advanced data visualization using seaborn.

1. Explain , how simple line plot can be created using matplotlib? Show the adjustments done to the plot w.r.t line colors.

The simplest of all plots is the visualization of a single function y = f (x). Here we will create simple line plot.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np

#creating a figure and an axes
fig = plt.figure()
ax = plt.axes()

x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x))
#ax.plot function to plot some data
```



In Matplotlib, the ***figure*** (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels. The ***axes*** (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up the visualization.

Alternatively, we can use the pylab interface, which creates the figure and axes in the background. Ex: **plt.plot(x, np.sin(x))**

Adjusting the Plot: Line Colors

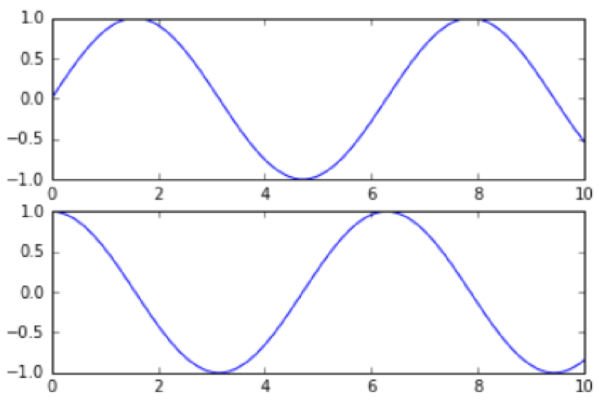
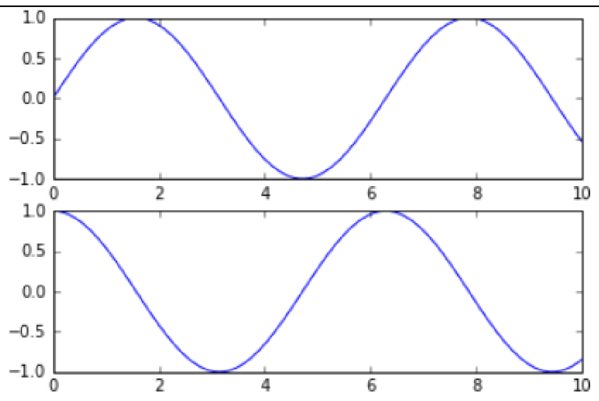
The plt.plot() function takes additional arguments that can be used to specify the color keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways.

```
plt.plot(x, np.sin(x - 0), color='blue') # specify color by name
plt.plot(x, np.sin(x - 1), color='g') # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44') # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 and 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

2. Distinguish MATLAB style and object-oriented interfaces of Matplotlib.

MATLAB style interface	**Object-oriented interface**
Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface.	The object-oriented interface is available for these more complicated situations, and for when we want more control over your figure.

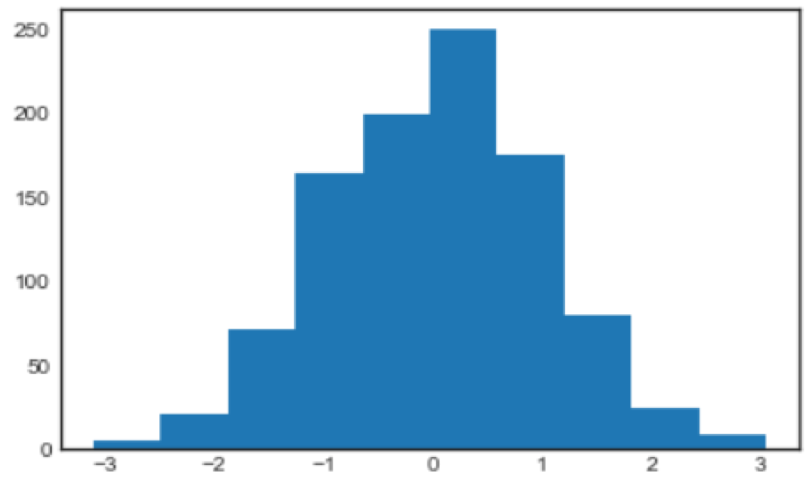
`plt.figure()  # create a plot figure`
 	
`# create the first of two panels and set current axis`
`plt.subplot(2, 1, 1) # (rows, columns, panel number)`
`plt.plot(x, np.sin(x))`
 	
`# create the second panel and set current axis`
`plt.subplot(2, 1, 2)`
`plt.plot(x, np.cos(x));`
	`# First create a grid of plots`
	`# ax will be an array of two Axes objects`
	`fig, ax = plt.subplots(2)`
	
	`# Call plot() method on the appropriate object`
	`ax[0].plot(x, np.sin(x))`
	`ax[1].plot(x, np.cos(x));`

	
Interface is stateful: it keeps track of the current" figure and axes, where all plt commands are applied. once the second panel is created, going back and adding something to the first is bit complex.	Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are methods of explicit Figure and Axes objects.

3. Write the lines of code to create a simple histogram using matplotlib library.

A simple histogram can be useful in understanding a dataset. the below code creates a simple histogram.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
data = np.random.randn(1000)
plt.hist(data)
```



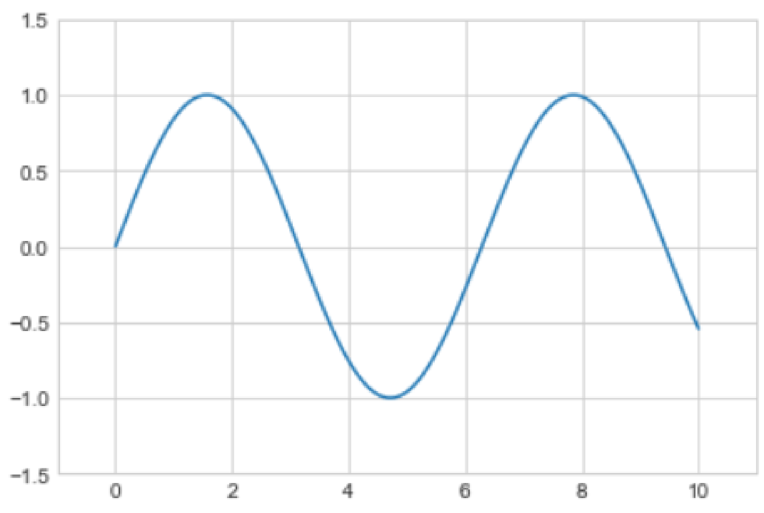
4. What are the two ways to adjust axis limits of the plot using Matplotlib? Explain with the example for each.

Matplotlib does a decent job of choosing default axes limits for your plot, but some- times it's nice to have finer control.

The two ways to adjust axis limits are:

- **using plt.xlim() and plt.ylim() methods**

```
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5)
```



- **using plt.axis()**

  The *plt.axis( )* method allows you to set the x and y limits with a single call, by passing a list that specifies [xmin, xmax, ymin, ymax].

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5])
```

5. List out the dissimilarities between plot() and scatter() functions while plotting scatter plot.

- The difference between the two functions is: with pyplot.plot() any property you apply (color, shape, size of points) will be applied across all points whereas in pyplot.scatter() you have more control in each point's appearance. That is, in plt.scatter() you can have the color, shape and size of each dot (datapoint) to vary based on another variable.

- While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, 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.

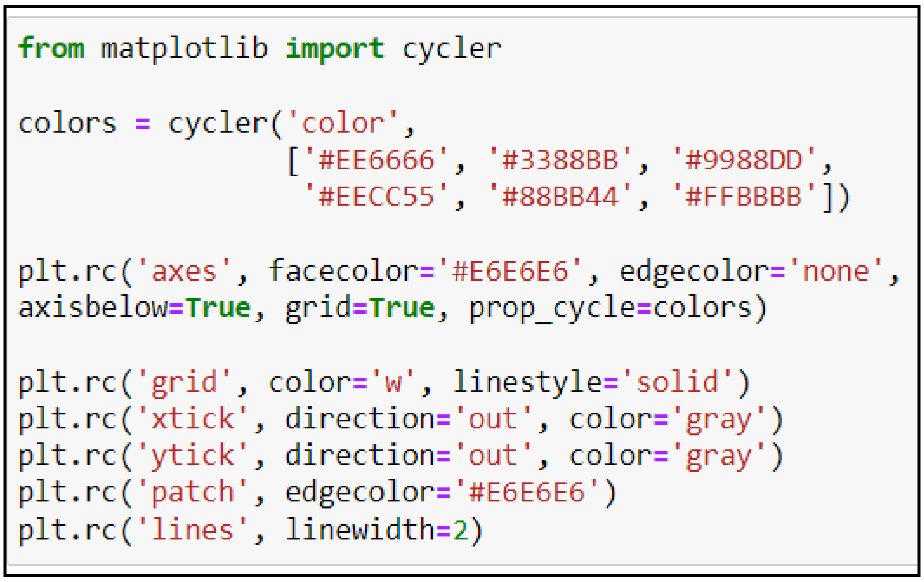
- For large datasets, the difference between these two can lead to vastly different performance, and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

6. How to customize the default plot settings of Matplotlib w.r.t runtime configuration and stylesheets? Explain with the suitable code snippet.

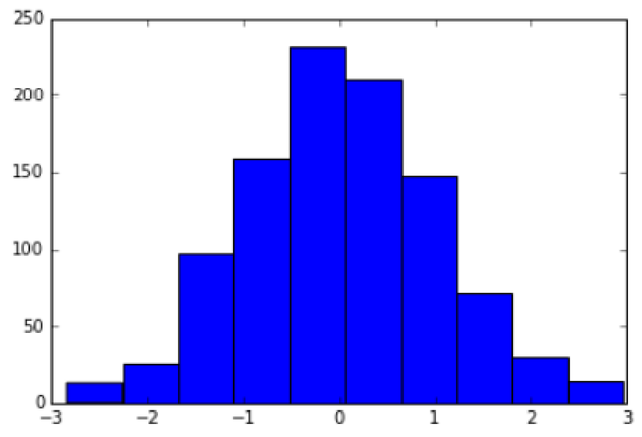
- Each time Matplotlib loads, it defines a runtime configuration (rc) containing the default styles for every plot element we create.
- We can adjust this configuration at any time using the **plt.rc** convenience routine.
- To modify the rc parameters, we'll start by saving a copy of the current rcParams dictionary, so we can easily reset these changes in the current session:

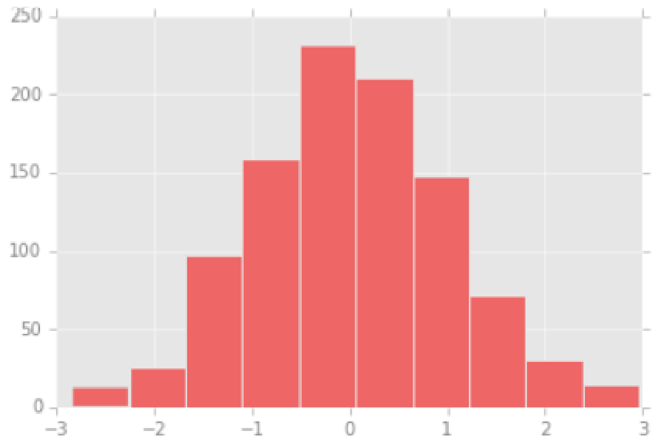
```
IPython_default = plt.rcParams.copy()
```

- Now we can use the plt.rc function to change some of these settings:



```
from matplotlib import cycler

colors = cycler('color',
                ['#EE6666', '#3388BB', '#9988DD',
                 '#EECC55', '#88BB44', '#FFBBBB'])

plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',
axisbelow=True, grid=True, prop_cycle=colors)

plt.rc('grid', color='w', linestyle='solid')
plt.rc('xtick', direction='out', color='gray')
plt.rc('ytick', direction='out', color='gray')
plt.rc('patch', edgecolor='#E6E6E6')
plt.rc('lines', linewidth=2)
```


A histogram in Matplotlib's default style


A customized histogram using rc settings

7. Elaborate on Seaborn versus Matplotlib with suitable examples.

Seaborn library is basically based on Matplotlib. Here is a detailed comparison between the two:

	Seaborn	**Matplotlib**
Functionality	Seaborn, on the other hand, provides a variety of visualization patterns. It uses fewer syntax and has easily interesting default themes. It specializes in statistics visualization and is used if one has to summarize data in visualizations and also show the distribution in the data.	Matplotlib is mainly deployed for basic plotting. Visualization using Matplotlib generally consists of bars, pies, lines, scatter plots and so on.

Handling Multiple Figures	Seaborn automates the creation of multiple figures. This sometimes leads to OOM (out of memory) issues.	Matplotlib has multiple figures can be opened, but need to be closed explicitly. plt.close() only closes the current figure. plt.close('all') would close them all.
Visualization	Seaborn is more integrated for working with Pandas data frames. It extends the Matplotlib library for creating beautiful graphics with Python using a more straightforward set of methods.	Matplotlib is a graphics package for data visualization in Python. It is well integrated with NumPy and Pandas. The pyplot module mirrors the MATLAB plotting commands closely. Hence, MATLAB users can easily transit to plotting with Python.
Data frames and Arrays	Seaborn works with the dataset as a whole and is much more intuitive than Matplotlib. For Seaborn, replot() is the entry API with 'kind' parameter to specify the type of plot which could be line, bar, or many of the other types. Seaborn is not stateful. Hence, plot() would require passing the object.	Matplotlib works with data frames and arrays. It has different stateful APIs for plotting. The figures and aces are represented by the object and therefore plot() like calls without parameters suffices, without having to manage parameters.
Flexibility	Seaborn avoids a ton of boilerplate by providing default themes which are commonly used.	Matplotlib is highly customizable and powerful.
Use Cases	Seaborn is for more specific use cases. Also, it is Matplotlib under the hood. It is specially meant for statistical plotting.	Pandas uses Matplotlib. It is a neat wrapper around Matplotlib.

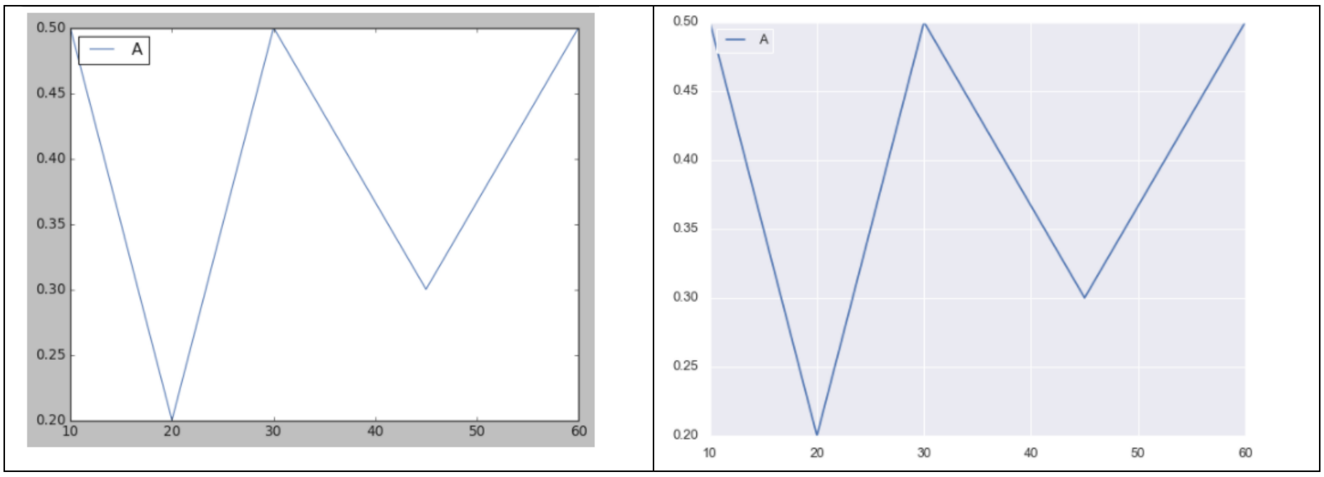
Let us assume

x=[10,20,30,45,60]

y=[0.5,0.2,0.5,0.3,0.5]

Matplotlib	**Seaborn**

#to plot the graph
`import matplotlib.pyplot as plt`
`plt.style.use('classic')`
`plt.plot(x, y)`
`plt.legend('ABCDEF',ncol=2, loc='upper left')`
	#to plot the graph
	`import seaborn as sns`
	`sns.set()`
	`plt.plot(x, y)`
	`plt.legend('ABCDEF',ncol=2, loc='upper left')`



8. List and describe different categories of colormaps with the suitable code snippets.

Three different categories of colormaps:

Sequential colormaps : These consist of one continuous sequence of colors (e.g., binary or viridis).

Divergent colormaps : These usually contain two distinct colors, which show positive and negative deviations from a mean (e.g., RdBu or PuOr).

Qualitative colormaps : These mix colors with no particular sequence (e.g., rainbow or jet).

```
speckles = (np.random.random(I.shape) < 0.01)
I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckles))
plt.figure(figsize=(10, 3.5))
plt.subplot(1, 2, 1)
plt.imshow(I, cmap='RdBu')
```

9. How to customize legends in the plot using matplotlib.

10. With the suitable example, describe how to draw histogram and kde plots using seaborn.

Often in statistical data visualization, all we want is to plot histograms and joint distributions of variables.