UNIT-5
Python for Data Visualization

Syllabus:

Python for Data Visualization: Visualization with matplotlib – box plot, line plots – scatter plots – visualizing errors – density and contour plots – histograms, binnings, and density – three-dimensional plotting – geographic data – data analysis using state models and seaborn – graph plotting using Plotly – interactive data visualization using Bokeh.

Visualization with Matplotlib

→ Matplotlib is a tool for visualization in Python.
→ Matplotlib is a multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.
→ It was conceived by John Hunter in2002
→ 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.

Two Interfaces of matplotlib:

→ A potentially confusing feature of Matplotlib is its dual interfaces:
- MATLAB-style interface
- Object-oriented interface.

MATLAB-style interface	Object-oriented interface
1. a convenient MATLAB-style state-based interface	a more powerful object-oriented interface.
2. The MATLAB-style tools are contained in the pyplot (plt) interface.	Tools are methods of explicit Figure and Axes object
3. depends 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
4. Best choice for more simple plots	Best choice for complicated plots
5. In MATLAB-style interface **plt.plot()** method is used	In the object-oriented interface to plotting **ax.plot()** method is used

6. Sample program:
import matplotlib.pyplot as plt
import numpy as np
# simple line plot via MATLAB-style interface 	
x = np.array([10,20,30,40,50])
y = np.array([100,200,300,400,500])
plt.plot(x, y)
plt.show()
	Sample program:
	import matplotlib.pyplot as plt
	import numpy as np
	# simple line plot via object-oriented interface
	x = np.array([10,20,30,40,50])
	y = np.array([100,200,300,400,500])
	fig = plt.figure()
	ax=plt.axes()
	ax.plot(x, y)
	plt.show()

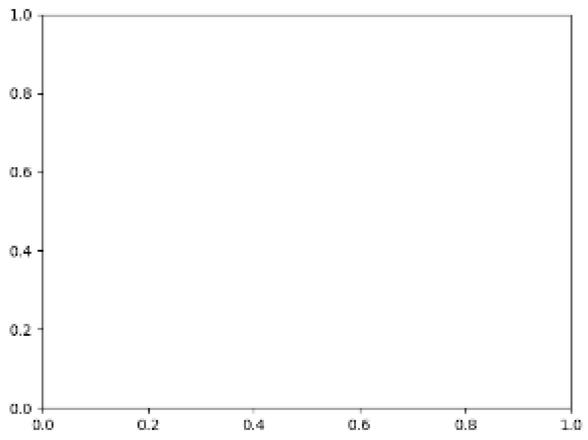
Simple Line Plots

→ The simplest of all plots is the visualization of a single function y = f(x)

→ For all Matplotlib plots, we start by creating a figure and an axes. In their simplest form, a figure and axes can be created as follows:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes()
```

Output:



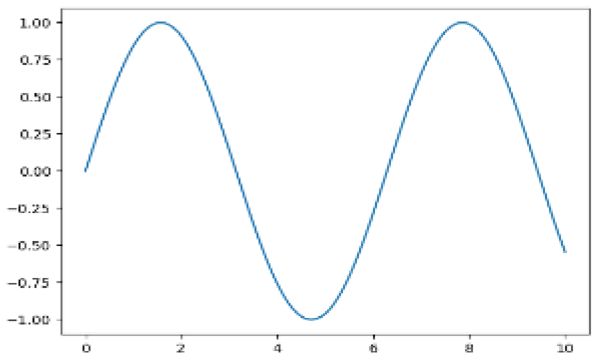
→ In Matplotlib, the *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels.

→ The *axes* (an instance of the class plt.Axes) are what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.

→ Example: Plotting a simple sinusoid via the object-oriented interface

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax=plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x))
plt.show()
```

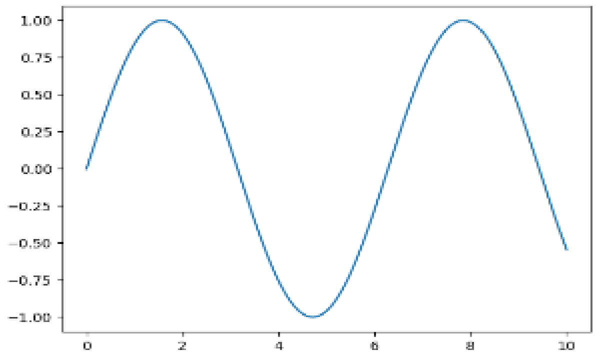
Output:



→ Example: Plotting a simple sinusoid via the MATLAB-style interface

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x))
plt.show()
```

Output:



Controlling the appearance of the axes and lines in plots

Adjusting the Plot:

- Line Colors and Styles
- Axes Limits

Labelling Plots

Adjusting the Plot: Line Colors and Styles

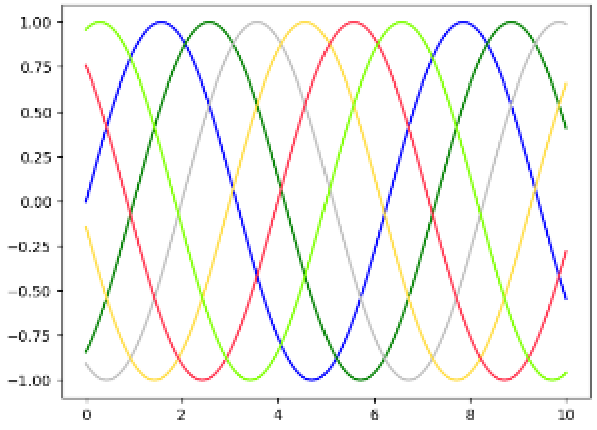
→ The plt.plot() function takes additional arguments that can be used to specify line colors and styles

→ To adjust the color, we can use the color keyword, which accepts a string argument representing virtually any imaginable color.

→ The color can be specified in a variety of ways :

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
# Controlling the color of plot elements
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
```

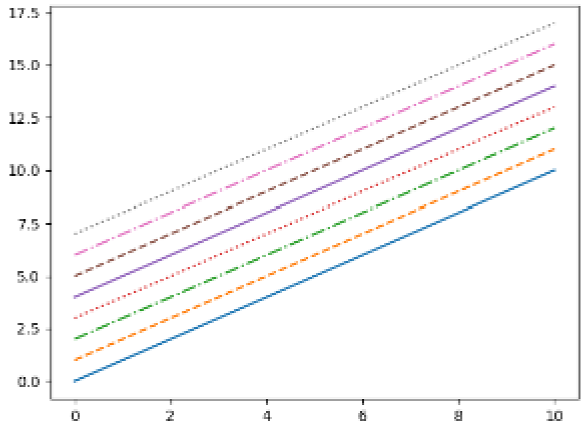
Output:



→ Similarly, we can adjust the line style using the linestyle keyword

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');
# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-') # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
 plt.show()
```

Output:

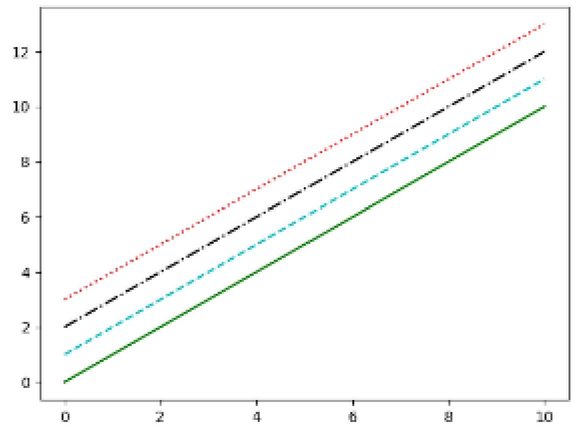


→ These linestyle and color codes can be combined into a single nonkeyword argument to the plt.plot() function.

```
import matplotlib.pyplot as plt
import numpy as np
```

```
x = np.linspace(0, 10, 1000)
plt.plot(x, x + 0, '-g') # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r'); # dotted red
plt.show()
```

Output:



Adjusting the Axes Limits:

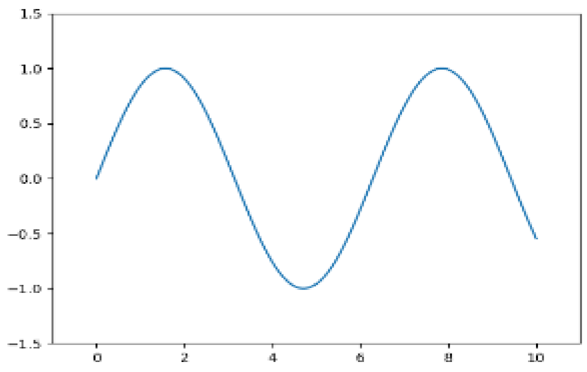
→ We can choose default axes limits for our plot, but sometimes it’s nice to have finer control.

→ The most basic way to adjust axis limits is to use the plt.xlim() and plt.ylim() methods

→ Sample program:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
plt.show()
```

Output:



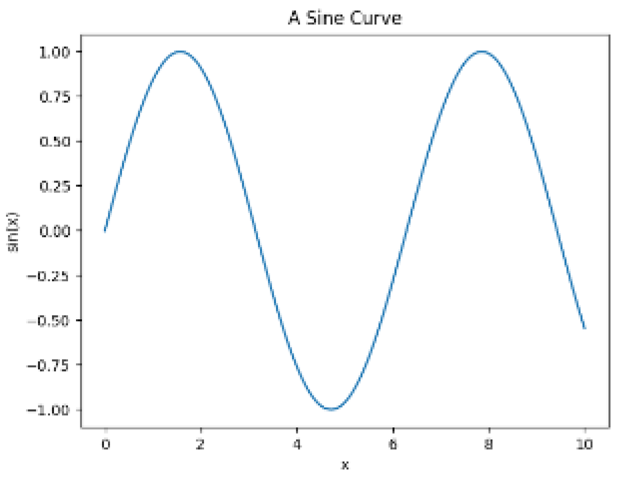
Labeling Plots

→ Labeling of plots includes titles, axis labels, and simple legends.

→ Titles and axis labels are the simplest such labels

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
plt.show()
```

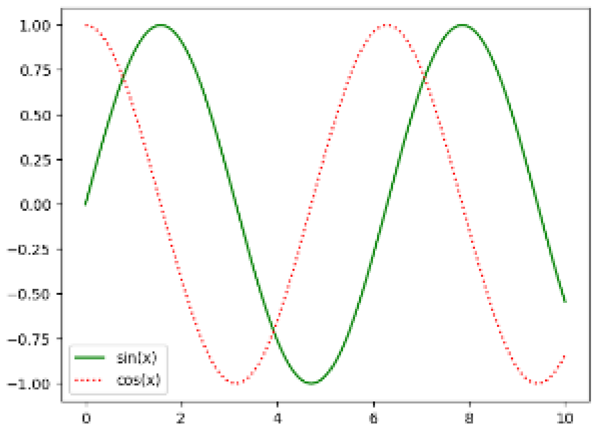
Output:



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

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':r', label='cos(x)')
plt.legend();
plt.show()
```

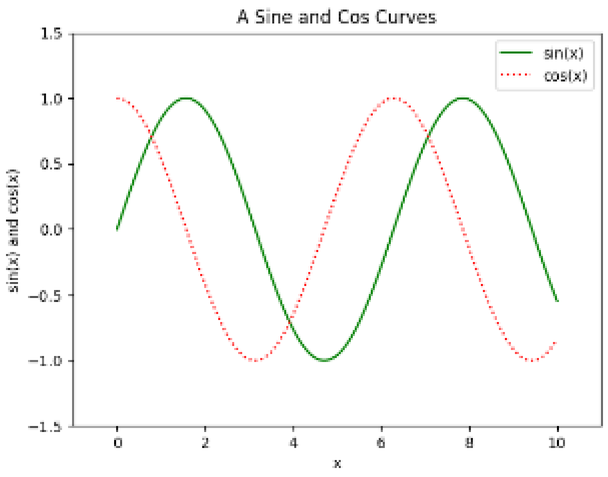
Output:



→ Python program for line plot that includes Line Colors and Styles, Axes Limits and Labelling Plots:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5)
plt.title("A Sine and Cos Curves")
plt.xlabel("x")
plt.ylabel("sin(x) and cos(x)")
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.linestyle='solid'
plt.plot(x, np.cos(x), ':r',label='cos(x)')
plt.linestyle='dotted'
plt.legend()
plt.show()
```

Output:



Simple Scatter Plots

→ Another commonly used plot type is the simple scatter plot, a close cousin of the line plot.

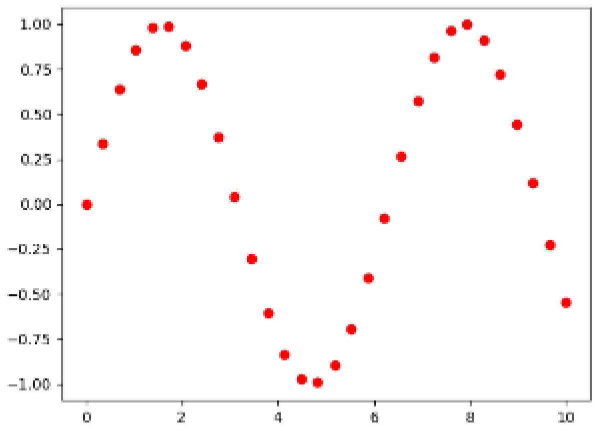
→ Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

Scatter Plots with plt.plot

→ Sample program:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='red');
plt.show()
```

→ Output:



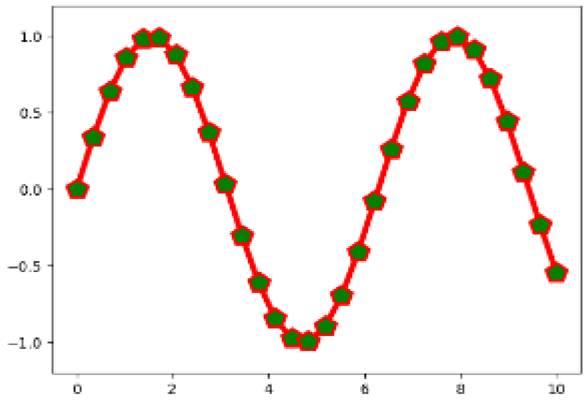
→ The third argument in the function call plt.plot is a character that represents the type of symbol used for the plotting.

→ Additional keyword arguments to plt.plot specify a wide range of properties of the lines and markers.

→ Example program:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, '-p', color='red',
markersize=15, linewidth=4,
markerfacecolor='green',
markeredgecolor='red',
markeredgewidth=2)
plt.ylim(-1.2, 1.2);
plt.show()
```

Output:



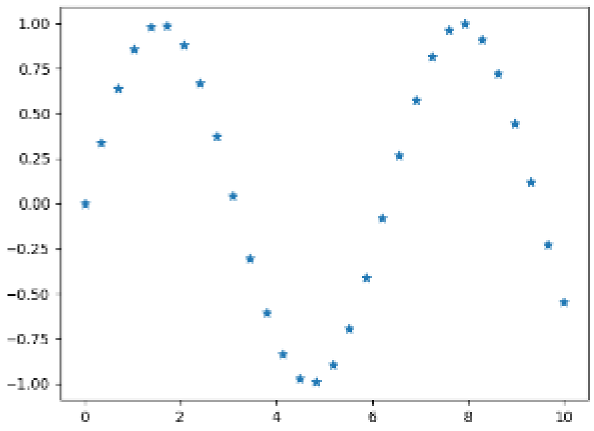
Scatter Plots with plt.scatter

→ A second, more powerful method of creating scatter plots is the plt.scatter function, which can be used very similarly to the plt.plot function

→ Sample program:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.scatter(x, y, marker='*');
plt.show()
```

Output:



plot Versus scatter:

→ 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.

→ 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 and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

Visualizing Errors

→ For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself.

→ In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

Basic Error bars

→ Error bars indicate how much each data point in a plot deviates from the actual value.

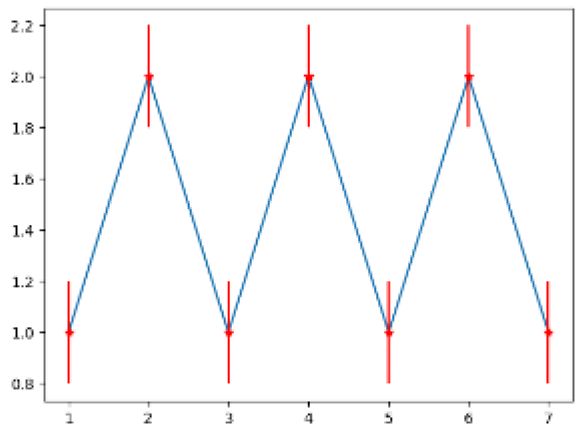
→ Error bars display the standard deviation of the distribution while the actual plot depicts the shape of the distribution.

→ For graphs present in the research papers it is a necessary to display the error value along with the data points which depicts the amount of uncertainty present in the data.

→ The function matplotlib.pyplot.errorbar() plots given x values and y values in a graph and marks the error or standard deviation of the distribution on each point.

```
import matplotlib.pyplot as plt
import numpy as np
x =[1, 2, 3, 4, 5, 6, 7]
y =[1, 2, 1, 2, 1, 2, 1]
yerror = 0.2
plt.plot(x,y)
plt.errorbar(x, y, yerr=yerror, fmt='*', color='red');
plt.show()
```

Output:

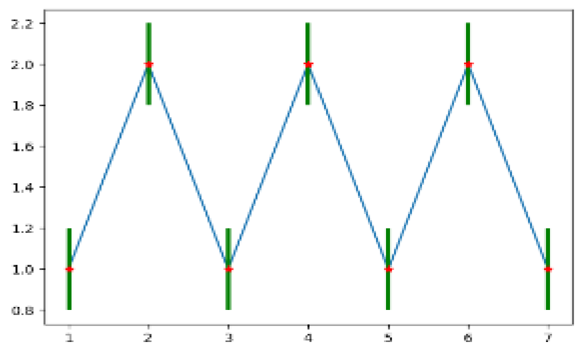


→ Here the fmt is a format code controlling the appearance of lines and points.

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

```
 import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
import numpy as np
x =[1, 2, 3, 4, 5, 6, 7]
y =[1, 2, 1, 2, 1, 2, 1]
yerror = 0.2
plt.plot(x,y)
plt.errorbar(x, y, yerr=yerror, fmt='*', color='red',ecolor='green',
elinewidth=3, capsize=0);
plt.show()
```

Output:



→ In addition to these options, you can also specify horizontal error bars (xerr), one sided error bars, and many other variants.

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.

Density and Contour Plots

(Matplotlib functions used to display three-dimensional data in two dimensions)

→ To display three-dimensional data in two dimensions using contours or color-coded regions.

→ There are three Matplotlib functions that can be helpful for this task:

- plt.contour for contour plots
- plt.contourf for filled contour plots
- plt.imshow for showing images.

Visualizing a Three-Dimensional Function

→ 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.

→ The way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays:

→ **Example:**

```
 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');
```

→ Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines.

→ Alternatively, we can color-code the lines by specifying a colormap with the cmap argument.

→ We'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range.

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

Histograms, Binnings, and Density

→ Histogram is the simple plot to represent the large data set. A histogram is a graph showing frequency distributions. It is a graph showing the number of observations within each given interval.

Parameters

- plt.hist( ) is used to plot histogram. The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument.
- bins - A histogram displays numerical data by grouping data into "bins" of equal width. Each bin is plotted as a bar whose height corresponds to how many data points are in that bin. Bins are also sometimes called "intervals", "classes", or "buckets".
- normed - Histogram normalization is a technique to distribute the frequencies of the histogram over a wider range than the current range.
- x - (n,) array or sequence of (n,) arrays Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.
- histtype - {'bar', 'barstacked', 'step', 'stepfilled'}, optional The type of histogram to draw.
    - 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
    - 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
    - 'step' generates a lineplot that is by default unfilled.

    - 'stepfilled' generates a lineplot that is by default filled.

  Default is 'bar'

- align - {'left', 'mid', 'right'}, optional

  Controls how the histogram is plotted.
    - 'left': bars are centered on the left bin edges.
    - 'mid': bars are centered between the bin edges.
    - 'right': bars are centered on the right bin edges.

  Default is 'mid'
- orientation - {'horizontal', 'vertical'}, optional

  If 'horizontal', barh will be used for bar-type histograms and the bottom kwarg will be the left edges.
- color - color or array_like of colors or None, optional

  Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

  Default is None
- label - str or None, optional. Default is None

Other parameter

- **kwargs - Patch properties, it allows us to pass a variable number of keyword arguments to a python function. ** denotes this type of function.

  Example

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

→ The hist() function has many options to tune both the calculation and the display; here's an example of a more customized histogram.

```
plt.hist(data, bins=30, alpha=0.5,histtype='stepfilled',color='steelblue',
         edgecolor='none');
```

→ The plt.hist docstring has more information on other customization options available. I find this combination of histtype='stepfilled' along with some transparency alpha to be very useful when comparing histograms of several distributions

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)
kwargs = dict(histtype='stepfilled', alpha=0.3, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

Two-Dimensional Histograms and Binnings

- → We can create histograms in two dimensions by dividing points among two dimensional bins.
- → We would define x and y values. Here for example We'll start by defining some data—an x and y array drawn from a multivariate Gaussian distribution:
- → Simple way to plot a two-dimensional histogram is to use Matplotlib's plt.hist2d() function

  Example

  ```
  mean = [0, 0]
  cov = [[1, 1], [1, 2]]
  x, y = np.random.multivariate_normal(mean, cov, 1000).T
  plt.hist2d(x, y, bins=30, cmap='Blues')
  cb = plt.colorbar()
  cb.set_label('counts in bin')
  ```

Kernel density estimation

- → Another common method of evaluating densities in multiple dimensions is kernel density estimation (KDE).
- → KDE can be thought of as a way to "smear out" the points in space and add up the result to obtain a smooth function.
- → KDE has a smoothing length that effectively slides the knob between detail and smoothness

Three-Dimensional Plotting

- → Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization.
- → We enable three-dimensional plots by importing the mplot3d toolkit

  ```
  from mpl_toolkits import mplot3d
  ```

- → We can create a three-dimensional axes by passing the keyword projection='3d' to any of the normal axes creation routines:

  ```
  import numpy as np
  import matplotlib.pyplot as plt
  fig = plt.figure()
  ax = plt.axes(projection='3d')
  ```

- → Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook when running this code.

→ Different three-dimensional plots are
- Three-Dimensional Points and Lines
  (Three-dimensional Line or scatter Plot)
- Three-dimensional contour Plot
- Wireframe and Surface Plots
- Surface Triangulations (Triangulated surface plot)

Three-dimensional Line or scatter Plot

→ The most basic three-dimensional plot is a line or scatter plot created from sets of (x, y, z) triples.

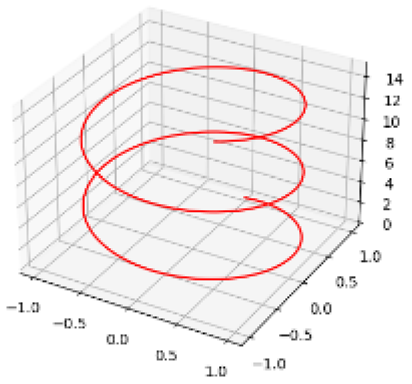
→ We can create these plots using the ax.plot3D and ax.scatter3D functions.

→ The call signature for these is nearly identical to that of their two-dimensional counterparts.

→ Example python program for three-dimensional Line Plot

```
 import matplotlib.pyplot as plt
 import numpy as np
from mpl_toolkits import mplot3d
fig = plt.figure()
# Data for a three-dimensional line
ax = plt.axes(projection='3d')
zline = np.linspace(0,15,1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline,yline,zline,'red')
plt.show()
```

Output:

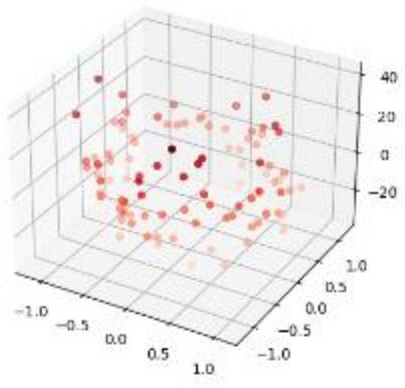


→ **Example program for three-dimensional scatter Plot:**

```
 import matplotlib.pyplot as plt
 import numpy as np
from mpl_toolkits import mplot3d
fig = plt.figure()
ax = plt.axes(projection='3d')
zdata = 15*np.random.randn(100)
xdata= np.sin(zdata) +0.1*np.random.randn(100)
```

```
ydata= np.cos(zdata) +0.1*np.random.randn(100)
ax.scatter3D(xdata,ydata,zdata,c=zdata,cmap='Reds')
plt.show()
```

Output:



Three-dimensional contour Plot:

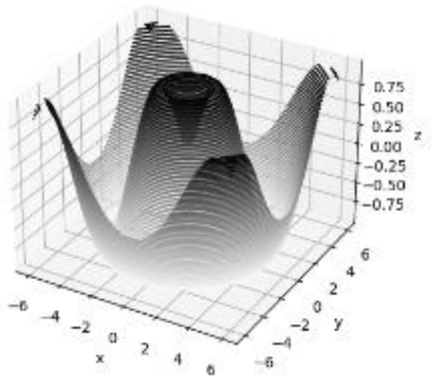
→ Analogous to the contour plots, mplot3d contains tools to create three-dimensional relief plots using the same inputs.

→ Like two-dimensional ax.contour plots, ax.contour3D requires all the input data to be in the form of two-dimensional regular grids, with the Z data evaluated at each point.

→ Python program

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d
def f(x, y):return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
ax = plt.axes(projection=’3d’)
ax.contour3D(X, Y, Z, 50, cmap=’binary’)
ax.set_xlabel(‘x’)
ax.set_ylabel(‘y’)
ax.set_zlabel(‘z’)
plt.show()
```

Output:



Wireframe and Surface Plots

→ These plots work on gridded data. These take a grid of values and project it onto the specified three-dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize.

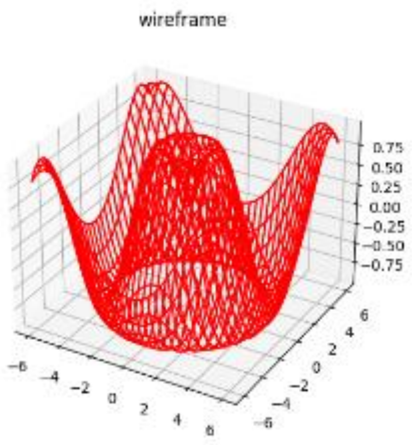
→ A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a color map to the filled polygons can aid perception of the topology of the surface being visualized.

→ Note that though the grid of values for a surface plot needs to be two-dimensional, it need not be rectilinear.

→ **Example for wireframe plot:**

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d
fig=plt.figure()
ax = plt.axes(projection='3d')
def f(x, y):return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
ax.plot_wireframe(X, Y, Z, color='red')
ax.set_title('wireframe');
plt.show()
```

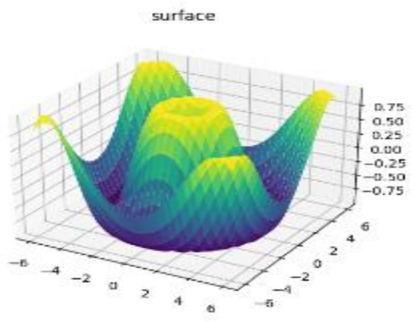
Output:



→ **Example for surface plot:**

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d
def f(x, y):return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none')
ax.set_title('surface');
plt.show()
```

Output:



Surface Triangulations

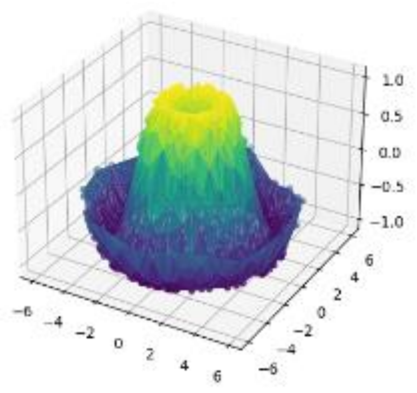
→ For some applications, the evenly sampled grids required by the preceding routines are overly restrictive and inconvenient. In these situations, the triangulation-based plots can be very useful.

→ Example program:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d
```

```
fig = plt.figure()
ax = plt.axes(projection='3d')
# Data for triangulat surface
def f(x, y):return np.sin(np.sqrt(x ** 2 + y ** 2))
theta = 2 * np.pi * np.random.random(1000)
r = 6 * np.random.random(1000)
x = np.ravel(r * np.sin(theta))
y = np.ravel(r * np.cos(theta))
z = f(x, y)
ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
ax.plot_trisurf(x, y, z,
cmap='viridis', edgecolor='none');
ax.set_title('A triangulated surface plot');
plt.show()
```

Output:



Geographic Data

→ One common type of visualization in data science is that of geographic data.

→ Matplotlib’s main tool for geographic data visualization is the Basemap toolkit, which is one of several Matplotlib toolkits that live under the mpl_toolkits namespace.

→ Basemap feels a bit clunky to use, and often even simple visualizations take much longer to render

→ Basemap is a useful tool for Python users to have in their virtual toolbelts.

Map Projections

→ Map projections are used to project a spherical map, such as that of the Earth, onto a flat surface without somehow distorting it or breaking its continuity.

→ Depending on the intended use of the map projection, there are certain map features (e.g., direction, area, distance, shape, or other considerations)
that are useful to maintain.

→ The Basemap package implements several projections, all referenced by a short format code. Some of the more common ones are:
  - Cylindrical projections
  - Pseudo-cylindrical projections
  - Perspective projections
  - Conic projections

Cylindrical projection

→ The simplest of map projections are cylindrical projections, in which lines of constant latitude and longitude are mapped to horizontal and vertical lines, respectively.

→ This type of mapping represents equatorial regions quite well, but results in extreme distortions near the poles.

→ The spacing of latitude lines varies between different cylindrical projections, leading to different conservation properties, and different distortion near the poles.

→ Other cylindrical projections are the Mercator (projection='merc') and the cylindrical equal-area (projection='cea') projections.

→ The additional arguments to Basemap for this view specify the latitude (lat) and longitude (lon) of the lower-left corner (llcrnr) and upper-right corner (urcrnr) for the desired map, in units of degrees.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,  llcrnrlat=-90,
      urcrnrlat=90, llcrnrlon=-180, urcrnrlon=180, )
draw_map(m)
```

Pseudo-cylindrical projections

→ Pseudo-cylindrical projections relax the requirement that meridians (lines of constant longitude) remain vertical; this can give better properties near the poles of the projection.

→ The Mollweide projection (projection='moll') is one common example of this, in which all meridians are elliptical arcs

→ It is constructed so as to preserve area across the map: though there are distortions near the poles, the area of small patches reflects the true area.

→ Other pseudo-cylindrical projections are the sinusoidal (projection='sinu') and Robinson (projection='robin') projections.

→ The extra arguments to Basemap here refer to the central latitude (lat_0) and longitude (lon_0) for the desired map.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', resolution=None, lat_0=0, lon_0=0)
draw_map(m)
```

Perspective projections

→ Perspective projections are constructed using a particular choice of perspective point, similar to if you photographed the Earth from a particular point in space (a point which, for some projections, technically lies within the Earth!).

→ One common example is the orthographic projection (projection='ortho'), which shows one side of the globe as seen from a viewer at a very long distance. Thus, it can show only half the globe at a time.

→ Other perspective-based projections include the
- gnomonic projection (projection='gnom') and
- stereographic projection (projection='stere').

→ These are often the most useful for showing small portions of the map.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None, lat_0=50, lon_0=0)
draw_map(m);
```

Conic projections

→ A conic projection projects the map onto a single cone, which is then unrolled.

→ This can lead to very good local properties, but regions far from the focus point of the cone may become much distorted.

→ One example of this is the Lambert conformal conic projection (projection='lcc').

→ It projects the map onto a cone arranged in such a way that two standard parallels (specified in Basemap by lat_1 and lat_2) have well-represented distances, with scale decreasing between them and increasing outside of them.

→ Other useful conic projections are the equidistant conic (projection='eqdc') and the Albers equal-area (projection='aea') projection

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None, lon_0=0, lat_0=50,
      lat_1=45, lat_2=55, width=1.6E7, height=1.2E7)
draw_map(m)
```

Drawing a Map Background

→ The Basemap package contains a range of useful functions for drawing borders of physical features like continents, oceans, lakes, and rivers, as well as political boundaries such as countries and US states and counties.

→ The following are some of the available drawing functions :

- Physical boundaries and bodies of water
  drawcoastlines() - Draw continental coast lines
  drawlsmask() - Draw a mask between the land and sea, for use with projecting images on one or the other
  drawmapboundary() - Draw the map boundary, including the fill color for oceans
  drawrivers() - Draw rivers on the map
  fillcontinents() - Fill the continents with a given color; optionally fill lakes with another color
- Political boundaries
  drawcountries() - Draw country boundaries
  drawstates() - Draw US state boundaries
  drawcounties() - Draw US county boundaries
- Map features
  drawgreatcircle() - Draw a great circle between two points
  drawparallels() - Draw lines of constant latitude
  drawmeridians() - Draw lines of constant longitude
  drawmapscale() - Draw a linear scale on the map
- Whole-globe images
  bluemarble() - Project NASA's blue marble image onto the map
  shadedrelief() - Project a shaded relief image onto the map
  etopo() - Draw an etopo relief image onto the map
  warpimage() - Project a user-provided image onto the map

Plotting Data on Maps

→ The Basemap toolkit is the ability to over-plot a variety of data onto a map background.

→ There are many map-specific functions available as methods of the Basemap instance.

→ Some of these map-specific methods are:

contour()/contourf() - Draw contour lines or filled contours
imshow() - Draw an image
pcolor()/pcolormesh() - Draw a pseudocolor plot for irregular/regular meshes
plot() - Draw lines and/or markers
scatter() - Draw points with markers
quiver() - Draw vectors
barbs() - Draw wind barbs
drawgreatcircle() - Draw a great circle

Data Analysis using Seaborn

→ Seaborn provides an API on top of Matplotlib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.

→ The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

→ Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output.

→ We can set the style by calling Seaborn's set() method. By convention, Seaborn is imported as sns

```
import seaborn as sns
sns.set()
```

→ Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with sns.kdeplot

```
import pandas as pd
import seaborn as sns
data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])
for col in 'xy':
sns.kdeplot(data[col], shade=True)
```

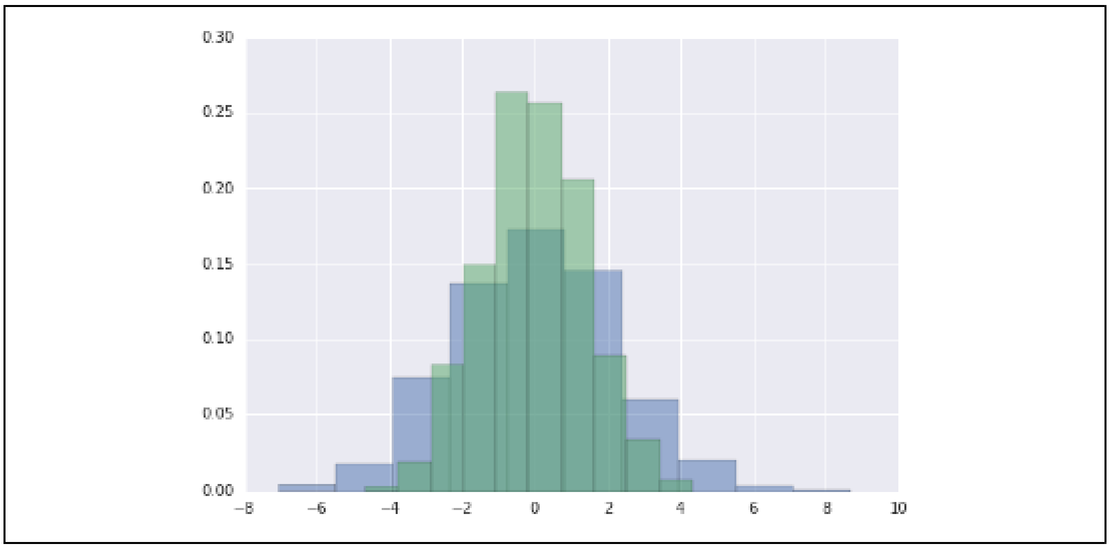


Figure 4-113. Histograms for visualizing distributions

→ Histograms and KDE can be combined using distplot

```
sns.distplot(data['x'])
sns.distplot(data['y']);
```

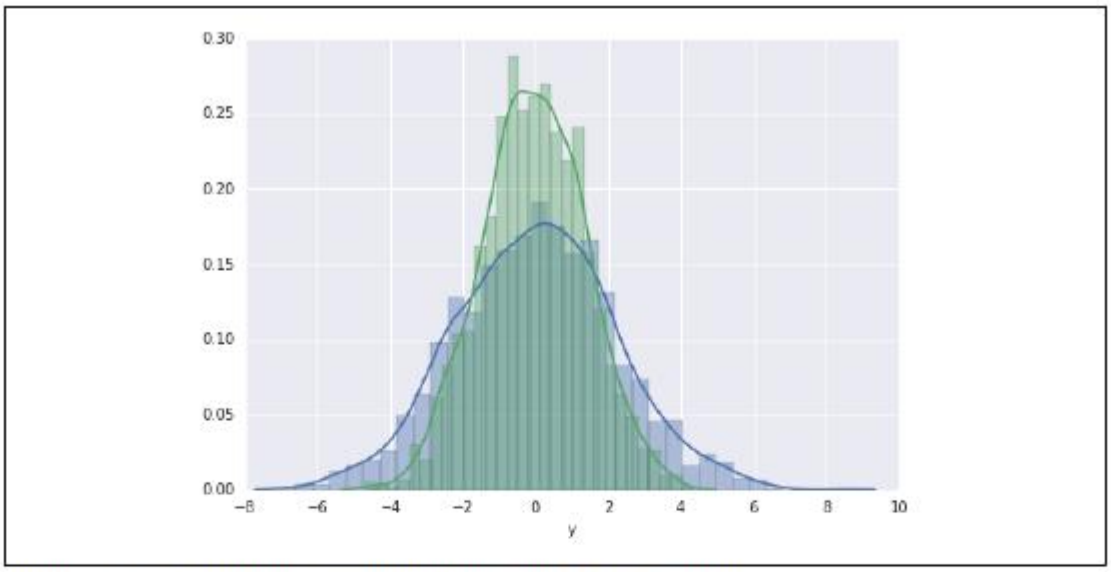


Figure 4-115. Kernel density and histograms plotted together

Data Analysis using StatsModels

→ StatsModels is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration.

→ StatsModels supports specifying models using R-style formulas and pandas DataFrames.

→ An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator.

→ Statsmodels is part of the Python scientific stack that is oriented towards data analysis, data science and statistics.

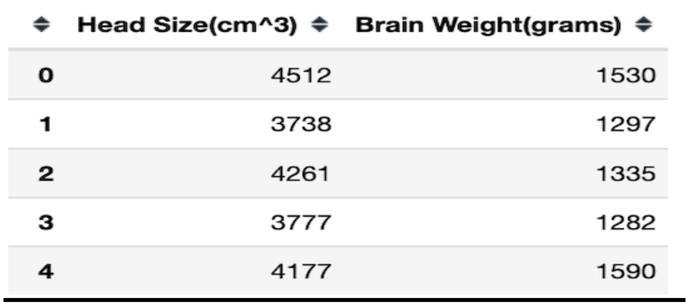
→ Statsmodels is built on top of the numerical libraries NumPy and SciPy, integrates with Pandas for data handling, and uses Patsy for an R-like formula interface.

→ Graphical functions are based on the Matplotlib library.

→ Statsmodels provides the statistical backend for other Python libraries.

→ Statsmodels is free software released under the Modified BSD (3-clause) license.

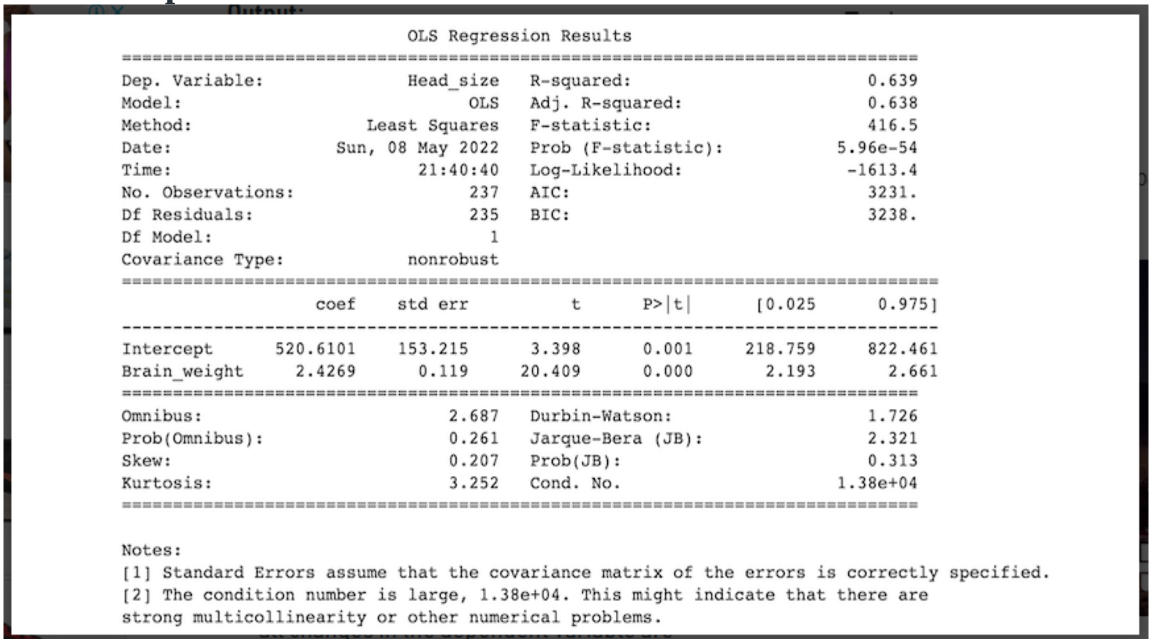
→ Sample program: Linear Regression in Python using Statsmodels



	Head Size(cm^3)	Brain Weight(grams)
0	4512	1530
1	3738	1297
2	4261	1335
3	3777	1282
4	4177	1590

```
# import packages
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

# loading the csv file
df = pd.read_csv('headbrain1.csv')
print(df.head())

# fitting the model
df.columns = ['Head_size', 'Brain_weight']
model = smf.ols(formula='Head_size ~ Brain_weight', data=df).fit()

# model summary
print(model.summary())
```

Output:



```
                            OLS Regression Results
==============================================================================
Dep. Variable:              Head_size   R-squared:                       0.639
Model:                            OLS   Adj. R-squared:                  0.638
Method:                 Least Squares   F-statistic:                     416.5
Date:                Sun, 08 May 2022   Prob (F-statistic):           5.96e-54
Time:                        21:40:40   Log-Likelihood:                -1613.4
No. Observations:                 237   AIC:                             3231.
Df Residuals:                     235   BIC:                             3238.
Df Model:                           1
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept      520.6101    153.215      3.398      0.001     218.759     822.461
Brain_weight     2.4269      0.119     20.409      0.000       2.193       2.661
==============================================================================
Omnibus:                        2.687   Durbin-Watson:                   1.726
Prob(Omnibus):                  0.261   Jarque-Bera (JB):                2.321
Skew:                           0.207   Prob(JB):                        0.313
Kurtosis:                       3.252   Cond. No.                     1.38e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.38e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Graph Plotting using Plotly

- → **Python Plotly** Library is an open-source library that can be used for data visualization and understanding data simply and easily.
- → Plotly supports various types of plots like line charts, scatter plots, histograms, box plots, etc.
- → Plotly has hover tool capabilities that allow us to detect any outliers or anomalies in a large number of data points.
- → It is visually attractive that can be accepted by a wide range of audiences.
- → It allows us for the endless customization of our graphs that makes our plot more meaningful and understandable for others.

→ Plotly has some amazing features that make it preferable over other visualization tools or libraries which are:

- **Simplicity:** The syntax is simple as each graph uses the same parameters.
- **Interactivity:** This allows users to interact with graphs on display, allowing for a better storytelling experience. Zooming in and out, point value display, panning graphs, and hovering over data values allow us to spot outliers or anomalies in massive numbers of sample points.
- **Attractivity:** The visuals are very attractive and can be accepted by a wide range of audiences.
- **Customizability:** It allows endless customization of graphs which makes plots more meaningful and understandable for others. It allows users to personalize graphs.

Modules of Plotly

1. Plotly Express: The plotly express module is used to produce professional and easy plots.

   import plotly.express as px

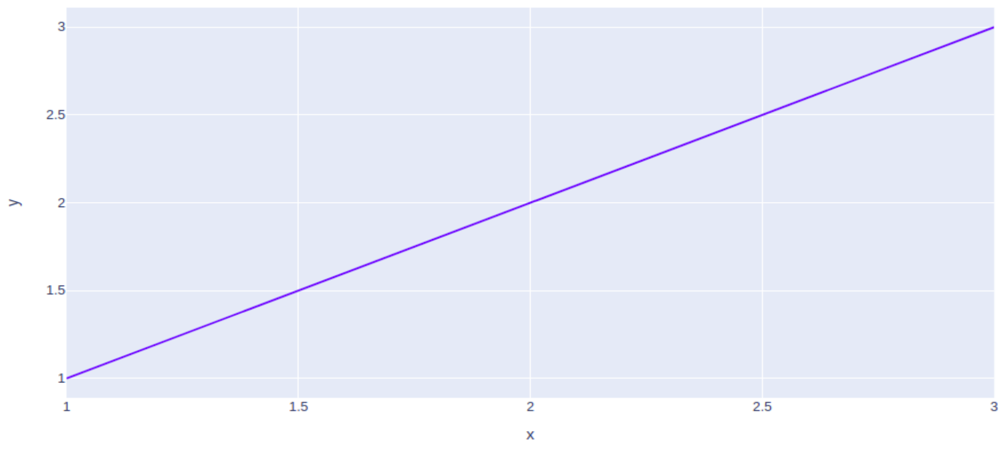
2. Plotly Graph Objects: This is used to produce more complex plots than that of plotly express .

   import plotly.graph_objects as go

Sample program:

```
import plotly.express as px

# Creating the Figure instance
fig = px.line(x=[1, 2, 3], y=[1, 2, 3])

# showing the plot
fig.show()
```

Output:



Interactive data visualization using Bokeh

→ ***Bokeh*** is a data visualization library in Python that provides high-performance interactive charts and plots.

→ Bokeh output can be obtained in various mediums like notebook, html and server. It is possible to embed bokeh plots in Django and flask apps.

→ Bokeh is an interactive visualization library in python. The best feature which bokeh provides is highly interactive graphs and plots that target modern web browsers for presentations.

→ Bokeh helps us to make elegant, and concise charts with a wide range of various charts.

→ Bokeh primarily focuses on converting the data source into JSON format which then uses as input for BokehJS.

→ Some of the best features of Bokeh are:

- **Flexibility:** Bokeh provides simple charts and customs charts too for complex use-cases.
- **Productivity:** Bokeh has an easily compatible nature and can work with Pandas and Jupyter notebooks.
- **Styling:** We have control of our chart and we can easily modify the charts by using custom Javascript.
- **Open source:** Bokeh provide a large number of examples and idea to start with and it is distributed under Berkeley Source Distribution (BSD) license.

→ With bokeh, we can easily visualize large data and create different charts in an attractive and elegant manner.

Benefits of Bokeh:

→ Bokeh allows to build complex statistical plots quickly and through simple commands

→ Bokeh provides output in various medium like html, notebook and server

→ We can also embed Bokeh visualization to flask and django app

→ Bokeh can transform visualization written in other libraries like matplotlib, seaborn, ggplot

→ Bokeh has flexibility for applying interaction, layouts and different styling option to visualization.

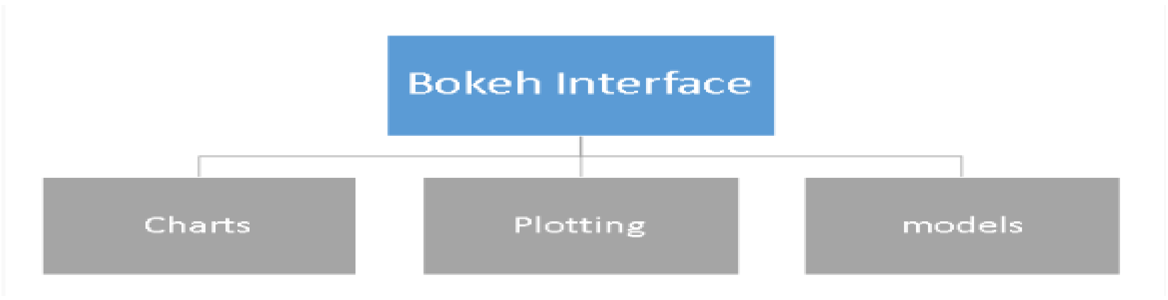
→ Bokeh provides two visualization interfaces to users:

- **bokeh.models** : A low level interface that provides high flexibility to application developers.
- **bokeh.plotting** : A high level interface for creating visual glyphs.

Visualization with Bokeh

→ Bokeh offers both powerful and flexible features which imparts simplicity and highly advanced customization.

→ It provides multiple visualization interfaces to the user as shown below:

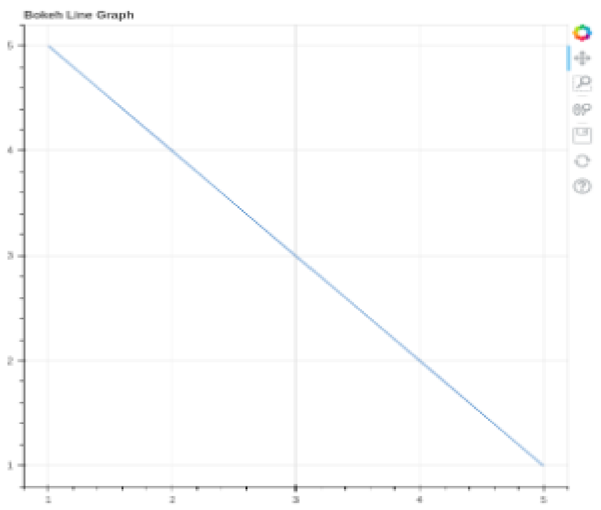


- **Charts:** a *high-level* interface that is used to build complex statistical plots as quickly and in a simplistic manner.
- **Plotting:** an *intermediate-level* interface that is centered around composing visual glyphs.
- **Models:** a *low-level* interface that provides the maximum flexibility to application developers.

→ Sample Program:

```
# importing the modules
from bokeh.plotting import figure, output_file, show

# instantiating the figure object
graph = figure(title = "Bokeh Line Graph")

# the points to be plotted
x = [1, 2, 3, 4, 5]
y = [5, 4, 3, 2, 1]

# plotting the line graph
graph.line(x, y)

# displaying the model
show(graph)
```

Output:



Tutorial Questions:

1. What is matplotlib? Specify the two interfaces used by it with example python code
2. Write python program to plot line plot by assuming your own data and explain the various attributes of line plot
3. Write python program to visualize the your own data using scatter plot and explain its parameters
4. Demonstrate the creating of line plot and scatter plot in matplotlib using different parameters with corresponding python code.
5. Briefly explain about geographic data with basemap with example programs.
   (or)Demonstrate different common map projections with example programs
   (or) Explain in details about the functions of mpl-toolkit for geographic data visualization
6. Elaborate the usage of histogram for data exploration and explain its attributes.
7. Demonstrate Matplotlib functions that can be helpful to display three-dimensional data in two-dimensions
8. Explore Seaborn plots with example programs
9. Elaborate error visualization methods in pyplot
10. Showcase three dimensional drawing in matplotlib with corresponding python code
    (or) Discuss in detail about the three- dimensional plotting of matplot module.
11. Appraise the following i)Histogram ii) Binning iii) Density with appropriate python code
12. Outline the data analysis using StatsModels with example programs
13. Outline graph plotting using Plotly
14. Describe interactive data visualization using Bokeh