DATA SCIENCE

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data,[1][2] and apply knowledge and actionable insights from data across a broad range of application domains. Data science is related to data mining, machine learning and big data.

NUMPY

NumPy is a Python package. It stands for 'Numerical Python'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

Numeric, the ancestor of NumPy, was developed by Jim Hugunin. Another package Numarray was also developed, having some additional functionalities. In 2005, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package. There are many contributors to this open source project.

Operations using NumPy

Using NumPy, a developer can perform the following operations −

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.

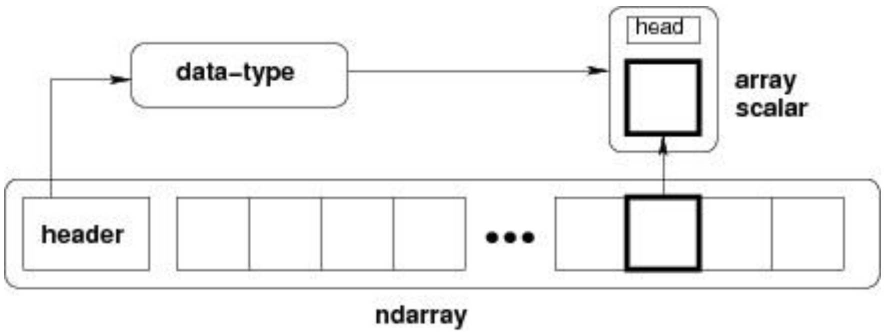
Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called **dtype**).

Ndarray Object

The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.

Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called **dtype**).

Any item extracted from ndarray object (by slicing) is represented by a Python object of one of array scalar types. The following diagram shows a relationship between ndarray, data type object (dtype) and array scalar type −

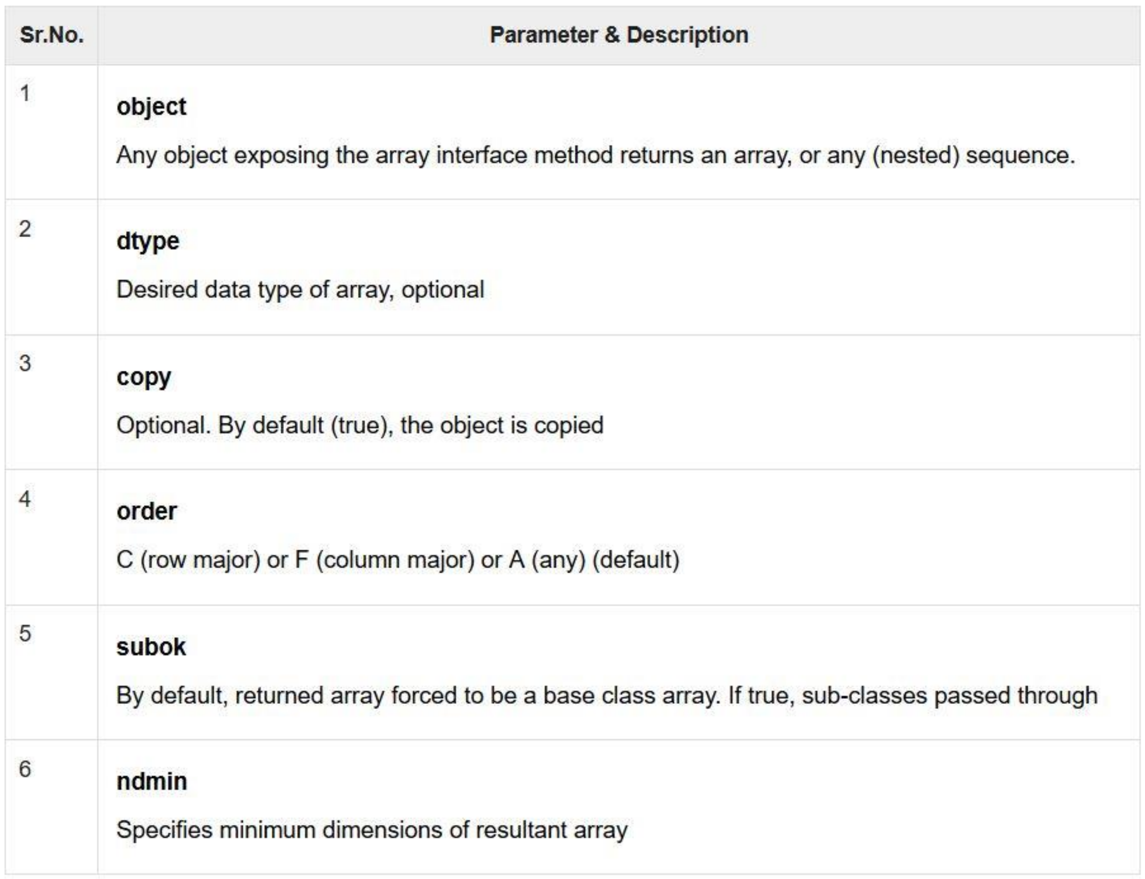


An instance of ndarray class can be constructed by different array creation routines described later in the tutorial. The basic ndarray is created using an array function in NumPy as follows −

```
numpy.array
```

It creates an ndarray from any object exposing array interface, or from any method that returns an array.

```
numpy.array(object, dtype = None, copy = True, order = None, subok = False,
ndmin = 0)
```



Sr.No.	Parameter & Description
1	**object** Any object exposing the array interface method returns an array, or any (nested) sequence.
2	**dtype** Desired data type of array, optional
3	**copy** Optional. By default (true), the object is copied
4	**order** C (row major) or F (column major) or A (any) (default)
5	**subok** By default, returned array forced to be a base class array. If true, sub-classes passed through
6	**ndmin** Specifies minimum dimensions of resultant array

KEY

We'll use shorthand in this cheat sheet

`arr` - A numpy Array object

IMPORTS

Import these to start

`import numpy as np`

IMPORTING/EXPORTING

`np.loadtxt('file.txt')` - From a text file

`np.genfromtxt('file.csv',delimiter=',')` - From a CSV file

`np.savetxt('file.txt',arr,delimiter=' ')` - Writes to a text file

`np.savetxt('file.csv',arr,delimiter=',')` - Writes to a CSV file

CREATING ARRAYS

`np.array([1,2,3])` - One dimensional array

`np.array([(1,2,3),(4,5,6)])` - Two dimensional array

`np.zeros(3)` - 1D array of length **3** all values **0**

`np.ones((3,4))` - **3**x**4** array with all values **1**

`np.eye(5)` - **5**x**5** array of **0** with **1** on diagonal (Identity matrix)

`np.linspace(0,100,6)` - Array of **6** evenly divided values from **0** to **100**

`np.arange(0,10,3)` - Array of values from **0** to less than **10** with step **3** (eg **[0,3,6,9]**)

`np.full((2,3),8)` - **2**x**3** array with all values **8**

`np.random.rand(4,5)` - **4**x**5** array of random floats between **0**-**1**

`np.random.rand(6,7)*100` - **6**x**7** array of random floats between **0**-**100**

`np.random.randint(5,size=(2,3))` - **2**x**3** array with random ints between **0**-**4**

INSPECTING PROPERTIES

`arr.size` - Returns number of elements in **arr**

`arr.shape` - Returns dimensions of **arr** (rows, columns)

`arr.dtype` - Returns type of elements in **arr**

`arr.astype(dtype)` - Convert **arr** elements to type **dtype**

`arr.tolist()` - Convert **arr** to a Python list

`np.info(np.eye)` - View documentation for **np.eye**

COPYING/SORTING/RESHAPING

`np.copy(arr)` - Copies **arr** to new memory

`arr.view(dtype)` - Creates view of **arr** elements with type **dtype**

`arr.sort()` - Sorts **arr**

`arr.sort(axis=0)` - Sorts specific axis of **arr**

`two_d_arr.flatten()` - Flattens 2D array **two_d_arr** to 1D

`arr.T` - Transposes **arr** (rows become columns and vice versa)

`arr.reshape(3,4)` - Reshapes **arr** to **3** rows, **4** columns without changing data

`arr.resize((5,6))` - Changes **arr** shape to **5**x**6** and fills new values with **0**

ADDING/REMOVING ELEMENTS

`np.append(arr,values)` - Appends **values** to end of **arr**

`np.insert(arr,2,values)` - Inserts **values** into **arr** before index **2**

`np.delete(arr,3,axis=0)` - Deletes row on index **3** of **arr**

`np.delete(arr,4,axis=1)` - Deletes column on index **4** of **arr**

COMBINING/SPLITTING

`np.concatenate((arr1,arr2),axis=0)` - Adds **arr2** as rows to the end of **arr1**

`np.concatenate((arr1,arr2),axis=1)` - Adds **arr2** as columns to end of **arr1**

`np.split(arr,3)` - Splits **arr** into **3** sub-arrays

`np.hsplit(arr,5)` - Splits **arr** horizontally on the **5**th index

INDEXING/SLICING/SUBSETTING

`arr[5]` - Returns the element at index **5**

`arr[2,5]` - Returns the 2D array element on index **[2][5]**

`arr[1]=4` - Assigns array element on index **1** the value **4**

`arr[1,3]=10` - Assigns array element on index **[1][3]** the value **10**

`arr[0:3]` - Returns the elements at indices **0,1,2** (On a 2D array: returns rows **0,1,2**)

`arr[0:3,4]` - Returns the elements on rows **0,1,2** at column **4**

`arr[:2]` - Returns the elements at indices **0,1** (On a 2D array: returns rows **0,1**)

`arr[:,1]` - Returns the elements at index **1** on all rows

`arr<5` - Returns an array with boolean values

`(arr1<3) & (arr2>5)` - Returns an array with boolean values

`~arr` - Inverts a boolean array

`arr[arr<5]` - Returns array elements smaller than **5**

SCALAR MATH

`np.add(arr,1)` - Add **1** to each array element

`np.subtract(arr,2)` - Subtract **2** from each array element

`np.multiply(arr,3)` - Multiply each array element by **3**

`np.divide(arr,4)` - Divide each array element by **4** (returns **np.nan** for division by zero)

`np.power(arr,5)` - Raise each array element to the **5**th power

VECTOR MATH

`np.add(arr1,arr2)` - Elementwise add **arr2** to **arr1**

`np.subtract(arr1,arr2)` - Elementwise subtract **arr2** from **arr1**

`np.multiply(arr1,arr2)` - Elementwise multiply **arr1** by **arr2**

`np.divide(arr1,arr2)` - Elementwise divide **arr1** by **arr2**

`np.power(arr1,arr2)` - Elementwise raise **arr1** raised to the power of **arr2**

`np.array_equal(arr1,arr2)` - Returns **True** if the arrays have the same elements and shape

`np.sqrt(arr)` - Square root of each element in the array

`np.sin(arr)` - Sine of each element in the array

`np.log(arr)` - Natural log of each element in the array

`np.abs(arr)` - Absolute value of each element in the array

`np.ceil(arr)` - Rounds up to the nearest int

`np.floor(arr)` - Rounds down to the nearest int

`np.round(arr)` - Rounds to the nearest int

STATISTICS

`np.mean(arr,axis=0)` - Returns mean along specific axis

`arr.sum()` - Returns sum of **arr**

`arr.min()` - Returns minimum value of **arr**

`arr.max(axis=0)` - Returns maximum value of specific axis

`np.var(arr)` - Returns the variance of array

`np.std(arr,axis=1)` - Returns the standard deviation of specific axis

`arr.corrcoef()` - Returns correlation coefficient of array

PANDAS

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.

Key Features of Pandas

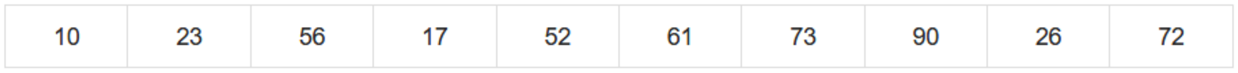
- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Pandas deals with the following three data structures −

- Series
- DataFrame
- Panel

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …



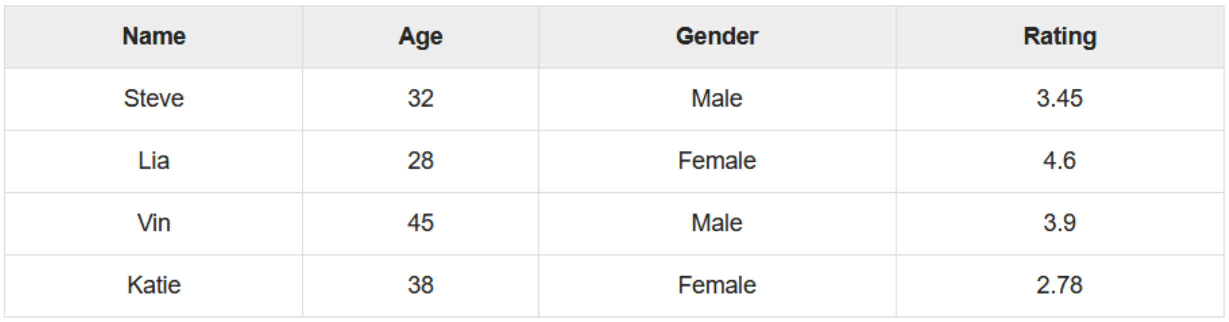
10	23	56	17	52	61	73	90	26	72

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

DataFrame

DataFrame is a two-dimensional array with heterogeneous data. For example,



Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Panel

Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Series Basic Functionality

Sr.No.	Attribute or Method & Description
1	**axes** Returns a list of the row axis labels
2	**dtype** Returns the dtype of the object.
3	**empty**

	Returns True if series is empty.
4	**ndim** Returns the number of dimensions of the underlying data, by definition 1.
5	**size** Returns the number of elements in the underlying data.
6	**values** Returns the Series as ndarray.
7	**head()** Returns the first n rows.
8	**tail()** Returns the last n rows.

DataFrame Basic Functionality

Let us now understand what DataFrame Basic Functionality is. The following tables lists down the important attributes or methods that help in DataFrame Basic Functionality.

Sr.No.	Attribute or Method & Description
1	**T** Transposes rows and columns.
2	**axes** Returns a list with the row axis labels and column axis labels as the only members.
3	**dtypes** Returns the dtypes in this object.

4	**empty** True if NDFrame is entirely empty [no items]; if any of the axes are of length 0.
5	**ndim** Number of axes / array dimensions.
6	**shape** Returns a tuple representing the dimensionality of the DataFrame.
7	**size** Number of elements in the NDFrame.
8	**values** Numpy representation of NDFrame.
9	**head()** Returns the first n rows.
10	**tail()** Returns last n rows.

DESCRIPTIVE STATISTICS

Functions & Description

Let us now understand the functions under Descriptive Statistics in Python Pandas. The following table list down the important functions −

Sr.No.	Function	Description
1	count()	Number of non-null observations

2	sum()	Sum of values
3	mean()	Mean of Values
4	median()	Median of Values
5	mode()	Mode of values
6	std()	Standard Deviation of the Values
7	min()	Minimum Value
8	max()	Maximum Value
9	abs()	Absolute Value
10	prod()	Product of Values
11	cumsum()	Cumulative Sum
12	cumprod()	Cumulative Product

KEY

We'll use shorthand in this cheat sheet

`df` - A pandas DataFrame object

`s` - A pandas Series object

IMPORTS

Import these to start

```
import pandas as pd
import numpy as np
```

IMPORTING DATA

`pd.read_csv(filename)` - From a CSV file

`pd.read_table(filename)` - From a delimited text file (like TSV)

`pd.read_excel(filename)` - From an Excel file

`pd.read_sql(query, connection_object)` - Reads from a SQL table/database

`pd.read_json(json_string)` - Reads from a JSON formatted string, URL or file.

`pd.read_html(url)` - Parses an html URL, string or file and extracts tables to a list of dataframes

`pd.read_clipboard()` - Takes the contents of your clipboard and passes it to `read_table()`

`pd.DataFrame(dict)` - From a dict, keys for columns names, values for data as lists

EXPORTING DATA

`df.to_csv(filename)` - Writes to a CSV file

`df.to_excel(filename)` - Writes to an Excel file

`df.to_sql(table_name, connection_object)` - Writes to a SQL table

`df.to_json(filename)` - Writes to a file in JSON format

`df.to_html(filename)` - Saves as an HTML table

`df.to_clipboard()` - Writes to the clipboard

CREATE TEST OBJECTS

Useful for testing

`pd.DataFrame(np.random.rand(20,5))` - 5 columns and 20 rows of random floats

`pd.Series(my_list)` - Creates a series from an iterable `my_list`

`df.index = pd.date_range('1900/1/30', periods=df.shape[0])` - Adds a date index

VIEWING/INSPECTING DATA

`df.head(n)` - First `n` rows of the DataFrame

`df.tail(n)` - Last `n` rows of the DataFrame

`df.shape()` - Number of rows and columns

`df.info()` - Index, Datatype and Memory information

`df.describe()` - Summary statistics for numerical columns

`s.value_counts(dropna=False)` - Views unique values and counts

`df.apply(pd.Series.value_counts)` - Unique values and counts for all columns

SELECTION

`df[col]` - Returns column with label `col` as Series

`df[[col1, col2]]` - Returns Columns as a new DataFrame

`s.iloc[0]` - Selection by position

`s.loc[0]` - Selection by index

`df.iloc[0,:]` - First row

`df.iloc[0,0]` - First element of first column

DATA CLEANING

`df.columns = ['a','b','c']` - Renames columns

`pd.isnull()` - Checks for null Values, Returns Boolean Array

`pd.notnull()` - Opposite of `s.isnull()`

`df.dropna()` - Drops all rows that contain null values

`df.dropna(axis=1)` - Drops all columns that contain null values

`df.dropna(axis=1,thresh=n)` - Drops all rows have have less than `n` non null values

`df.fillna(x)` - Replaces all null values with `x`

`s.fillna(s.mean())` - Replaces all null values with the mean (mean can be replaced with almost any function from the statistics section)

`s.astype(float)` - Converts the datatype of the series to float

`s.replace(1,'one')` - Replaces all values equal to `1` with `'one'`

`s.replace([1,3],['one','three'])` - Replaces all `1` with `'one'` and `3` with `'three'`

`df.rename(columns=lambda x: x + 1)` - Mass renaming of columns

`df.rename(columns={'old_name': 'new_name'})` - Selective renaming

`df.set_index('column_one')` - Changes the index

`df.rename(index=lambda x: x + 1)` - Mass renaming of index

FILTER, SORT, & GROUPBY

`df[df[col] > 0.5]` - Rows where the `col` column is greater than `0.5`

`df[(df[col] > 0.5) & (df[col] < 0.7)]` - Rows where `0.7 > col > 0.5`

`df.sort_values(col1)` - Sorts values by `col1` in ascending order

`df.sort_values(col2,ascending=False)` - Sorts values by `col2` in descending order

`df.sort_values([col1,col2], ascending=[True,False])` - Sorts values by `col1` in ascending order then `col2` in descending order

`df.groupby(col)` - Returns a groupby object for values from one column

`df.groupby([col1,col2])` - Returns a groupby object values from multiple columns

`df.groupby(col1)[col2].mean()` - Returns the mean of the values in `col2`, grouped by the values in `col1` (mean can be replaced with almost any function from the statistics section)

`df.pivot_table(index=col1,values=[col2,col3],aggfunc=mean)` - Creates a pivot table that groups by `col1` and calculates the mean of `col2` and `col3`

`df.groupby(col1).agg(np.mean)` - Finds the average across all columns for every unique column 1 group

`df.apply(np.mean)` - Applies a function across each column

`df.apply(np.max, axis=1)` - Applies a function across each row

JOIN/COMBINE

`df1.append(df2)` - Adds the rows in `df1` to the end of `df2` (columns should be identical)

`pd.concat([df1, df2],axis=1)` - Adds the columns in `df1` to the end of `df2` (rows should be identical)

`df1.join(df2,on=col1,how='inner')` - SQL-style joins the columns in `df1` with the columns on `df2` where the rows for `col` have identical values. `how` can be one of `'left'`, `'right'`, `'outer'`, `'inner'`

STATISTICS

These can all be applied to a series as well.

`df.describe()` - Summary statistics for numerical columns

`df.mean()` - Returns the mean of all columns

`df.corr()` - Returns the correlation between columns in a DataFrame

`df.count()` - Returns the number of non-null values in each DataFrame column

`df.max()` - Returns the highest value in each column

`df.min()` - Returns the lowest value in each column

`df.median()` - Returns the median of each column

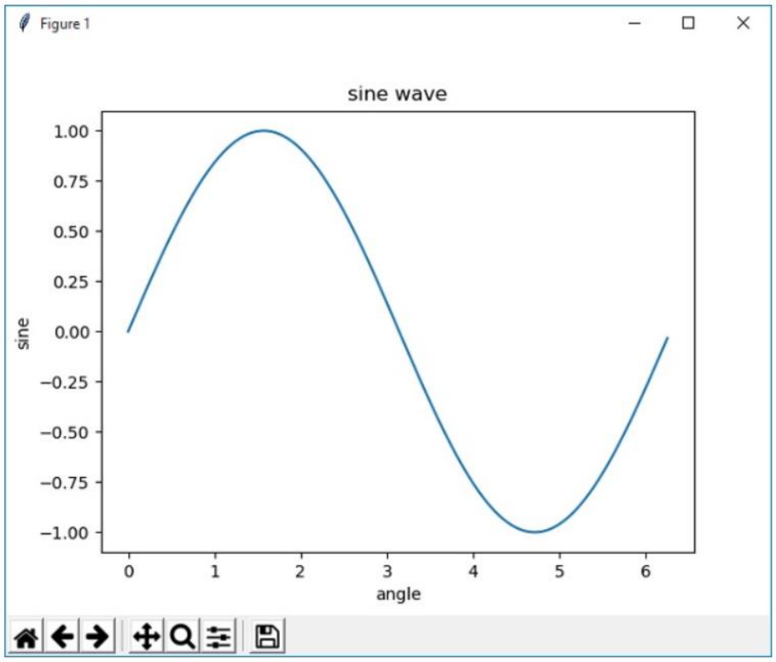
`df.std()` - Returns the standard deviation of each column

MATPLOTLIB

Matplotlib is one of the most popular Python packages used for data visualization. It is a cross-platform library for making 2D plots from data in arrays. Matplotlib is written in Python and makes use of NumPy, the numerical mathematics extension of Python. It provides an object-oriented API that helps in embedding plots in applications using Python GUI toolkits such as PyQt, WxPythonotTkinter. It can be used in Python and IPython shells, Jupyter notebook and web application servers also.

Simple plot

```
from matplotlib import pyplot as plt
import numpy as np
import math #needed for definition of pi
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)
plt.plot(x,y)
plt.xlabel("angle")
plt.ylabel("sine")
plt.title('sine wave')
plt.show()
```



Object oriented Interface

The main idea behind using the more formal object-oriented method is to create figure objects and then just call methods or attributes off of that object. This approach helps better in dealing with a canvas that has multiple plots on it.

In object-oriented interface, Pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses Pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

To begin with, we create a figure instance which provides an empty canvas.

```
fig = plt.figure()
```

Now add axes to figure. The **add_axes()** method requires a list object of 4 elements corresponding to left, bottom, width and height of the figure. Each number must be between 0 and 1 −

```
ax=fig.add_axes([0,0,1,1])
```

Set labels for x and y axis as well as title −

```
ax.set_title("sine wave")
ax.set_xlabel('angle')
ax.set_ylabel('sine')
```

Invoke the plot() method of the axes object.

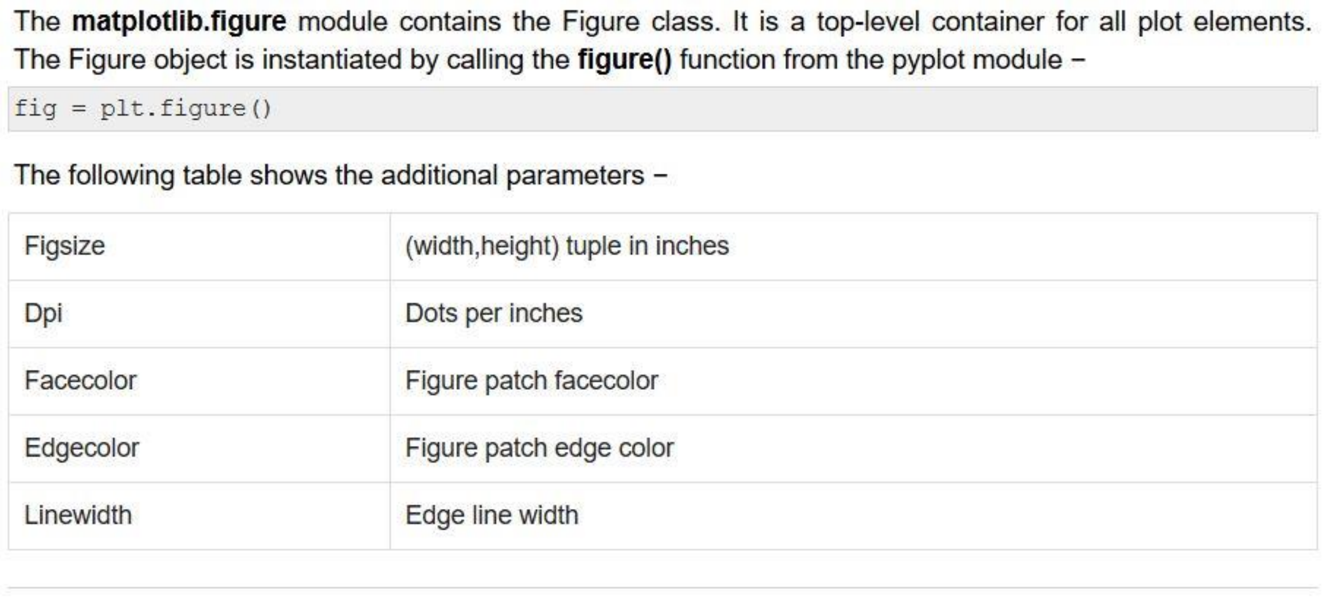
```
ax.plot(x,y)
```

If you are using Jupyter notebook, the %matplotlib inline directive has to be issued; the otherwistshow() function of pyplot module displays the plot.

Consider executing the following code −

```
from matplotlib import pyplot as plt
import numpy as np
import math
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.plot(x,y)
ax.set_title("sine wave")
ax.set_xlabel('angle')
ax.set_ylabel('sine')
plt.show()
```

MATPLOTLIB-FIGURE CLASS



The **matplotlib.figure** module contains the Figure class. It is a top-level container for all plot elements. The Figure object is instantiated by calling the **figure()** function from the pyplot module –

```
fig = plt.figure()
```

The following table shows the additional parameters –

Figsize	(width,height) tuple in inches
Dpi	Dots per inches
Facecolor	Figure patch facecolor
Edgecolor	Figure patch edge color
Linewidth	Edge line width

AXES CLASS

Axes object is the region of the image with the data space. A given figure can contain many Axes, but a given Axes object can only be in one Figure. The Axes contains two (or three in the case of 3D) Axis objects. The Axes class and its member functions are the primary entry point to working with the OO interface.

Axes object is added to figure by calling the add_axes() method. It returns the axes object and adds an axes at position rect [left, bottom, width, height] where all quantities are in fractions of figure width and height.

Parameter

Following is the parameter for the Axes class −

- rect − A 4-length sequence of [left, bottom, width, height] quantities.

```
ax=fig.add_axes([0,0,1,1])
```

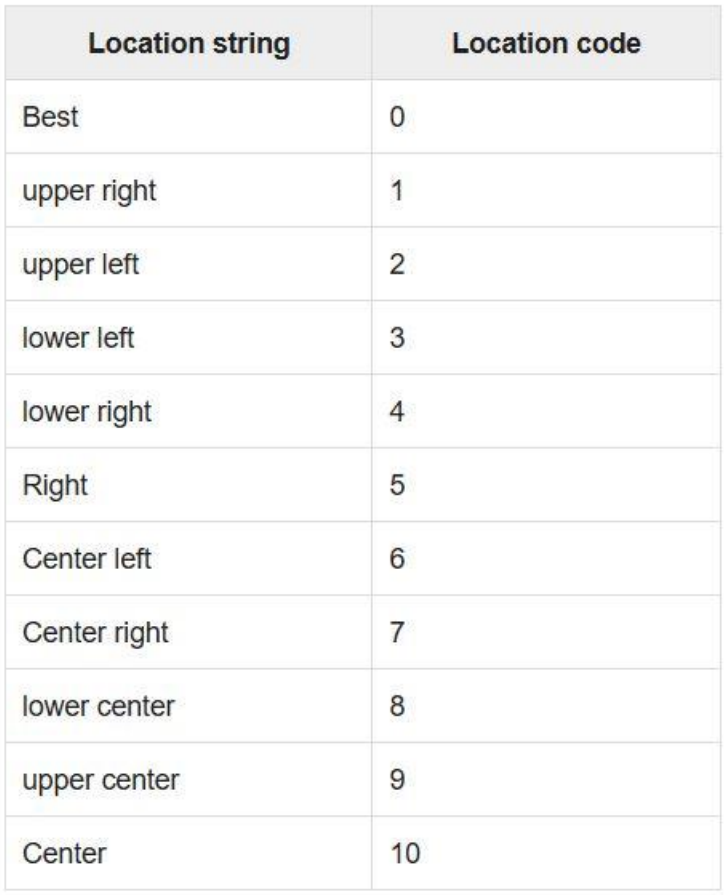
The following member functions of axes class add different elements to plot −

Legend

The **legend()** method of axes class adds a legend to the plot figure. It takes three parameters −

```
ax.legend(handles, labels, loc)
```

Where labels is a sequence of strings and handles a sequence of Line2D or Patch instances. loc can be a string or an integer specifying the legend location.



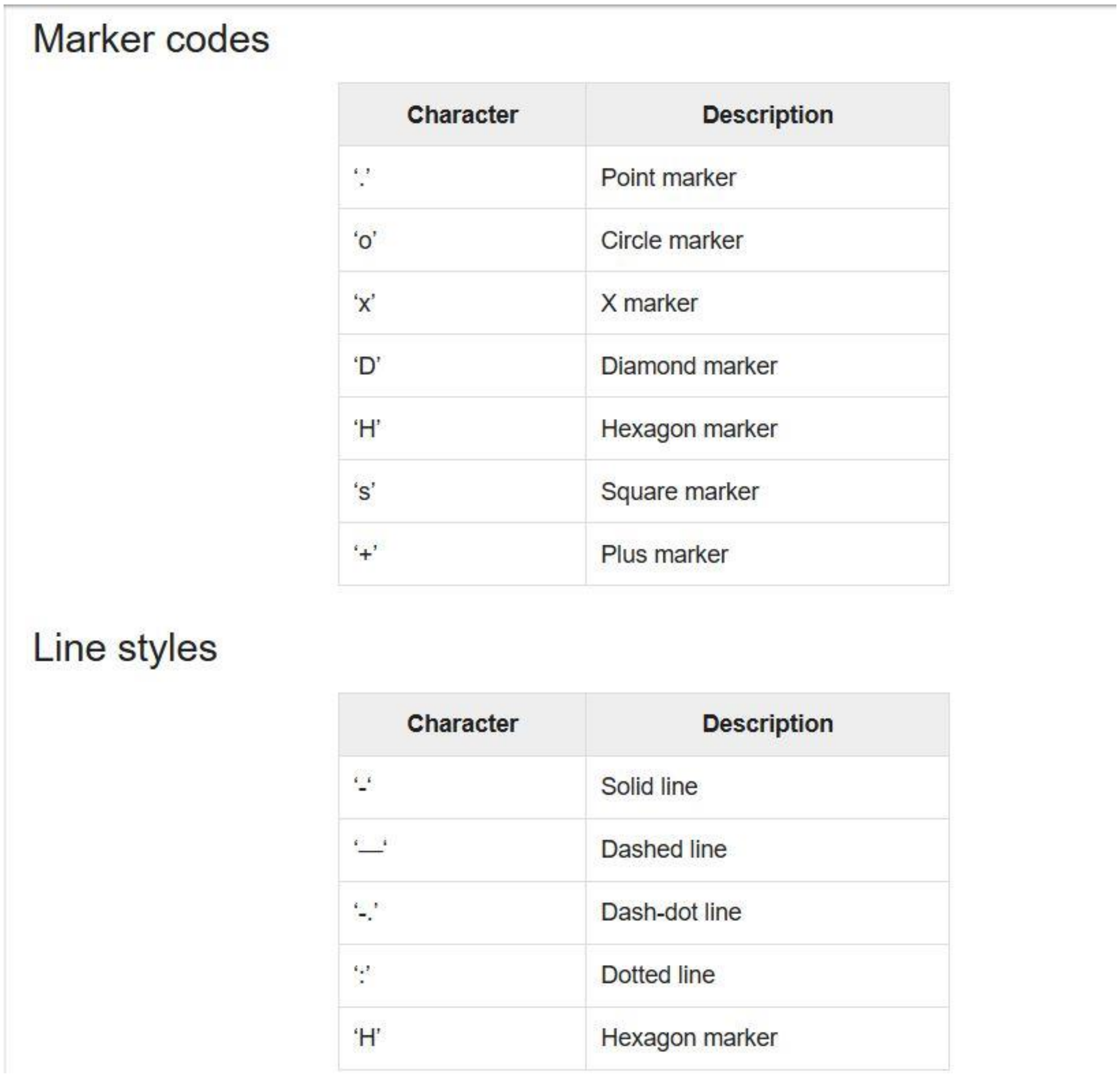
Location string	Location code
Best	0
upper right	1
upper left	2
lower left	3
lower right	4
Right	5
Center left	6
Center right	7
lower center	8
upper center	9
Center	10

axes.plot()

This is the basic method of axes class that plots values of one array versus another as lines or markers. The plot() method can have an optional format string argument to specify color, style and size of line and marker.

Color codes

Character	Color
‘b’	Blue
‘g’	Green
‘r’	Red
‘b’	Blue
‘c’	Cyan
‘m’	Magenta
‘y’	Yellow
‘k’	Black
‘b’	Blue
‘w’	White



Marker codes

Character	Description
‘.’	Point marker
‘o’	Circle marker
‘x’	X marker
‘D’	Diamond marker
‘H’	Hexagon marker
‘s’	Square marker
‘+’	Plus marker

Line styles

Character	Description
‘-‘	Solid line
‘—‘	Dashed line
‘-.’	Dash-dot line
‘:’	Dotted line
‘H’	Hexagon marker

MultiPlots

The **subplot()** function returns the axes object at a given grid position. The Call signature of this function is −

```
plt.subplot(subplot(nrows, ncols, index)
```

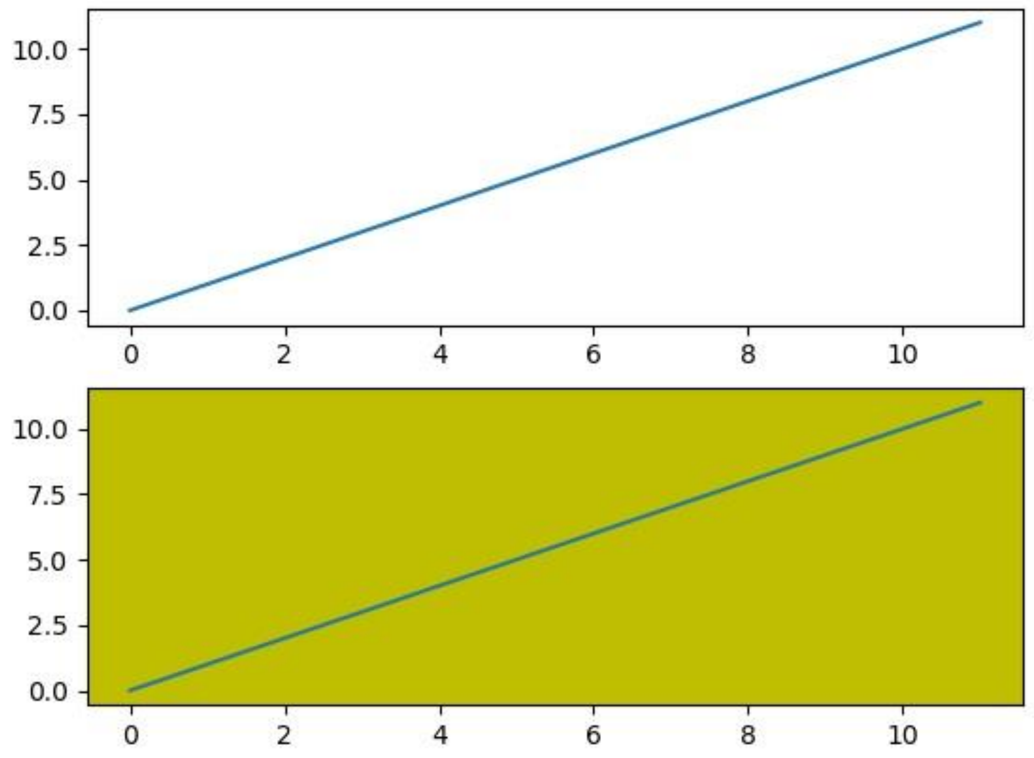
In the current figure, the function creates and returns an Axes object, at position index of a grid of nrows by ncolsaxes. Indexes go from 1 to nrows * ncols, incrementing in row-major order.Ifnrows, ncols and index are all less than 10. The indexes can also be given as single, concatenated, threedigitnumber.

For example, subplot(2, 3, 3) and subplot(233) both create an Axes at the top right corner of the current figure, occupying half of the figure height and a third of the figure width.

Creating a subplot will delete any pre-existing subplot that overlaps with it beyond sharing a boundary.

```
import matplotlib.pyplot as plt
# plot a line, implicitly creating a subplot(111)
plt.plot([1,2,3])
# now create a subplot which represents the top plot of a grid with 2 rows
and 1 column.
#Since this subplot will overlap the first, the plot (and its axes)
previously
created, will be removed
plt.subplot(211)
plt.plot(range(12))
plt.subplot(212, facecolor='y') # creates 2nd subplot with yellow background
plt.plot(range(12))
```

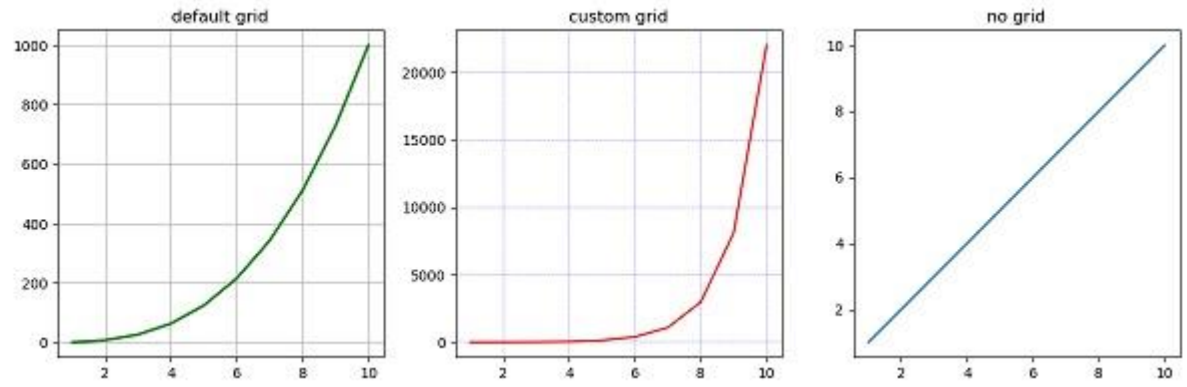
The above line of code generates the following output −



Grids

The grid() function of axes object sets visibility of grid inside the figure to on or off. You can also display major / minor (or both) ticks of the grid. Additionally color, linestyle and linewidth properties can be set in the grid() function.

```
import matplotlib.pyplot as plt
import numpy as np
fig, axes = plt.subplots(1,3, figsize = (12,4))
x = np.arange(1,11)
axes[0].plot(x, x**3, 'g',lw=2)
axes[0].grid(True)
axes[0].set_title('default grid')
axes[1].plot(x, np.exp(x), 'r')
axes[1].grid(color='b', ls = '-.', lw = 0.25)
axes[1].set_title('custom grid')
axes[2].plot(x,x)
axes[2].set_title('no grid')
fig.tight_layout()
plt.show()
```

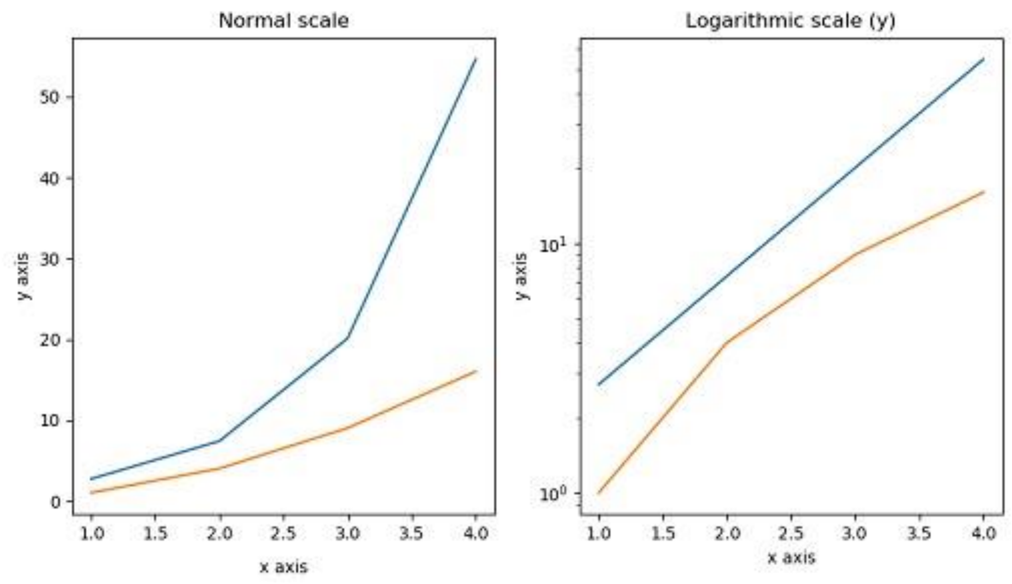


FORMATTING AXES

It is also required sometimes to show some additional distance between axis numbers and axis label. The labelpad property of either axis (x or y or both) can be set to the desired value.

Both the above features are demonstrated with the help of the following example. The subplot on the right has a logarithmic scale and one on left has its x axis having label at more distance.

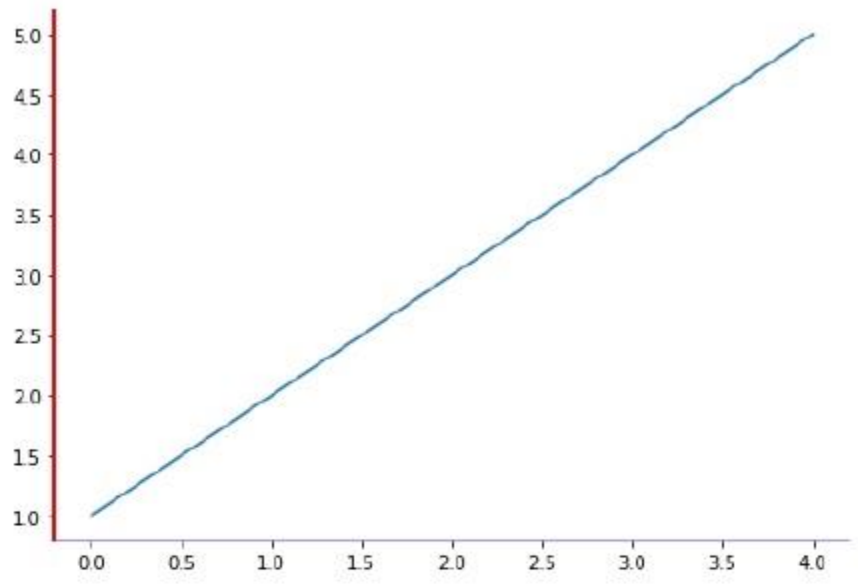
```
import matplotlib.pyplot as plt
import numpy as np
fig, axes = plt.subplots(1, 2, figsize=(10,4))
x = np.arange(1,5)
axes[0].plot( x, np.exp(x))
axes[0].plot(x,x**2)
axes[0].set_title("Normal scale")
axes[1].plot (x, np.exp(x))
axes[1].plot(x, x**2)
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)")
axes[0].set_xlabel("x axis")
axes[0].set_ylabel("y axis")
axes[0].xaxis.labelpad = 10
axes[1].set_xlabel("x axis")
axes[1].set_ylabel("y axis")
plt.show()
```



Axis spines are the lines connecting axis tick marks demarcating boundaries of plot area. The axes object has spines located at top, bottom, left and right.

Each spine can be formatted by specifying color and width. Any edge can be made invisible if its color is set to none.

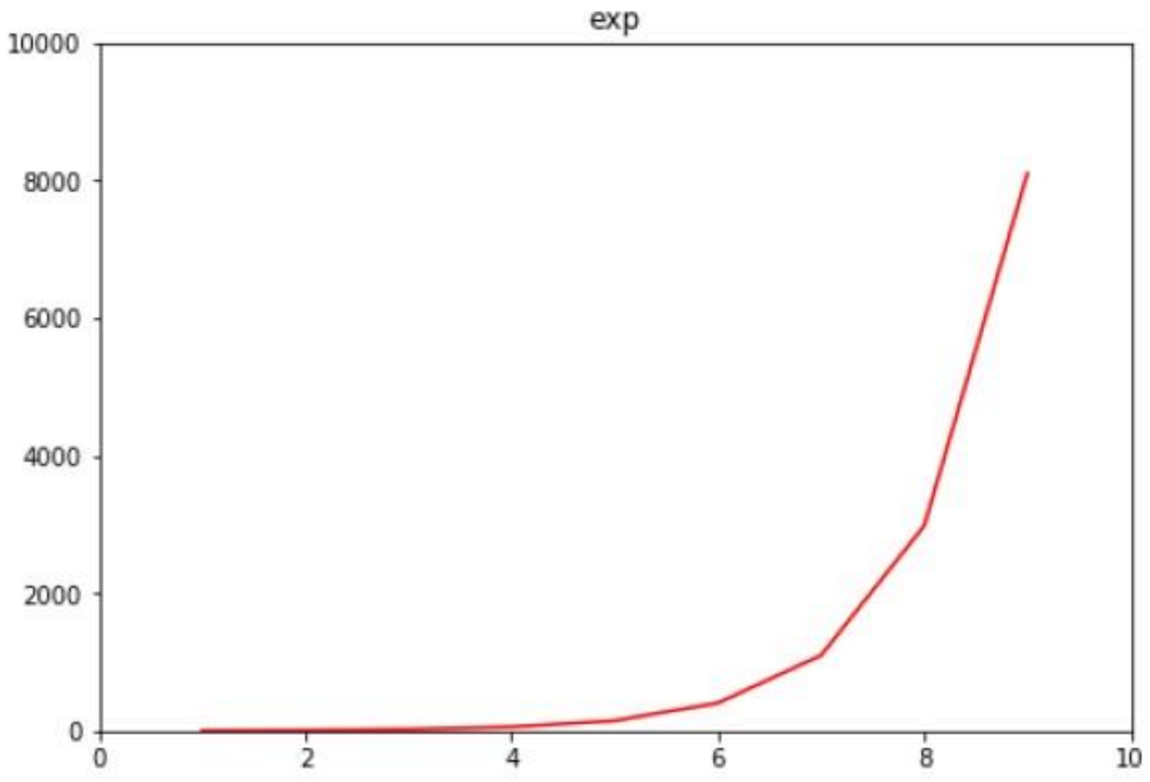
```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.spines['bottom'].set_color('blue')
ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)
ax.spines['right'].set_color(None)
ax.spines['top'].set_color(None)
ax.plot([1,2,3,4,5])
plt.show()
```



SETTING LIMITS

Matplotlib automatically arrives at the minimum and maximum values of variables to be displayed along x, y (and z axis in case of 3D plot) axes of a plot. However, it is possible to set the limits explicitly by using **set_xlim()** and **set_ylim()** functions.

```
import matplotlib.pyplot as plt
fig = plt.figure()
a1 = fig.add_axes([0,0,1,1])
import numpy as np
x = np.arange(1,10)
a1.plot(x, np.exp(x),'r')
a1.set_title('exp')
a1.set_ylim(0,10000)
a1.set_xlim(0,10)
plt.show()
```

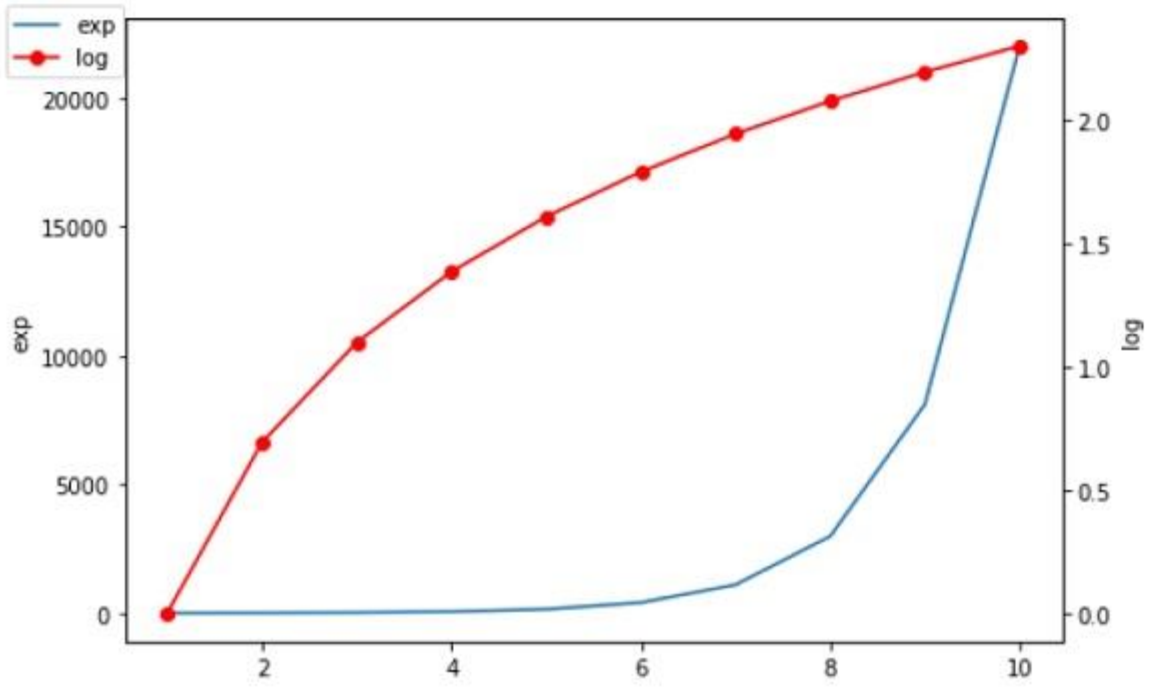


TWIN AXIS

It is considered useful to have dual x or y axes in a figure. Moreso, when plotting curves with different units together. Matplotlib supports this with the twinxand twiny functions.

In the following example, the plot has dual y axes, one showing exp(x) and the other showing log(x) −

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
a1 = fig.add_axes([0,0,1,1])
x = np.arange(1,11)
a1.plot(x,np.exp(x))
a1.set_ylabel('exp')
a2 = a1.twinx()
a2.plot(x, np.log(x),'ro-')
a2.set_ylabel('log')
fig.legend(labels = ('exp','log'),loc='upper left')
plt.show()
```



Bar Chart

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally.

A bar graph shows comparisons among discrete categories. One axis of the chart shows the specific categories being compared, and the other axis represents a measured value.

Matplotlib API provides the **bar()** function that can be used in the MATLAB style use as well as object oriented API.

The parameters to the function are −

x	sequence of scalars representing the x coordinates of the bars. align controls if x is the bar center (default) or left edge.
height	scalar or sequence of scalars representing the height(s) of the bars.
width	scalar or array-like, optional. the width(s) of the bars default 0.8
bottom	scalar or array-like, optional. the y coordinate(s) of the bars default None.
align	{‘center’, ‘edge’}, optional, default ‘center’

Scatter Plot

Scatter plots are used to plot data points on horizontal and vertical axis in the attempt to show how much one variable is affected by another. Each row in the data table is represented by a marker the position depends on its values in the columns set on the X and Y axes. A third variable can be set to correspond to the color or size of the markers, thus adding yet another dimension to the plot.

Histogram

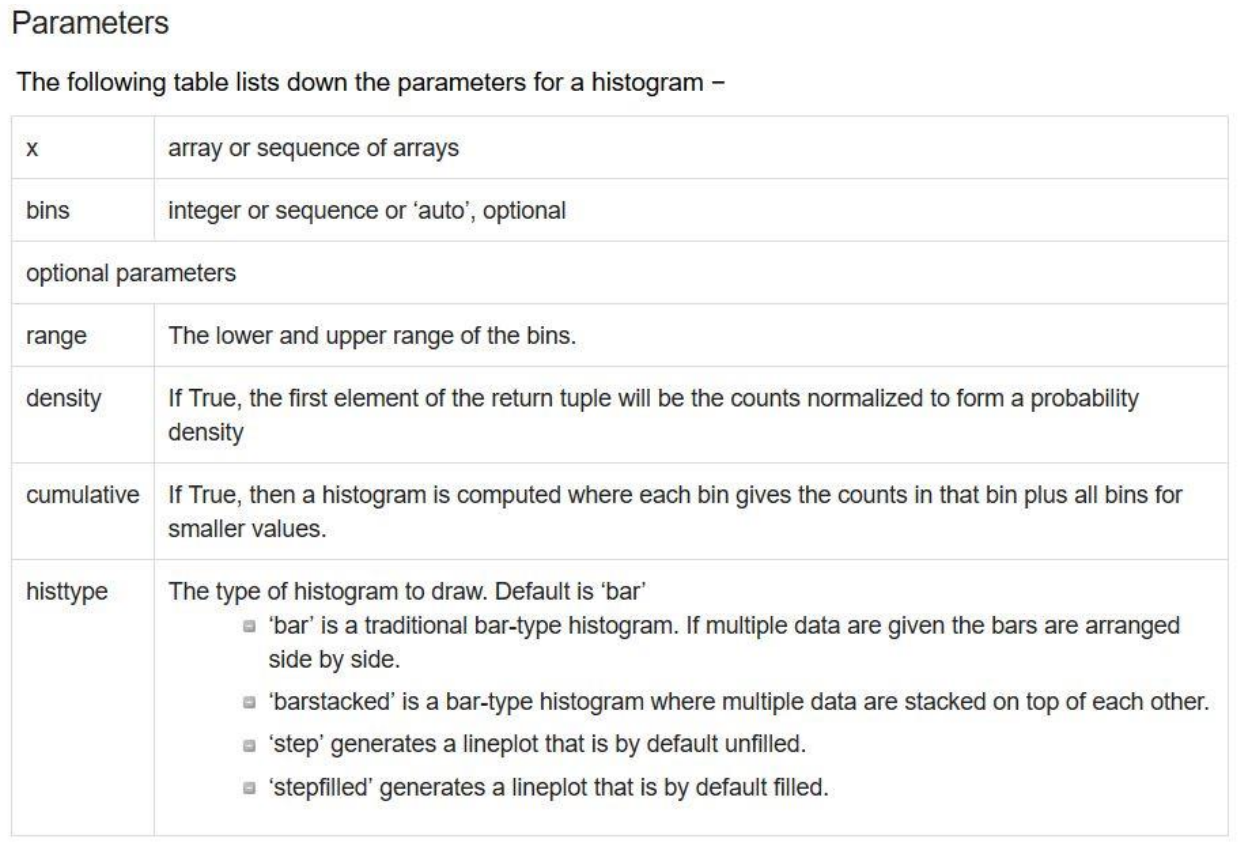
A histogram is an accurate representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable. It is a kind of bar graph.

To construct a histogram, follow these steps −

- **Bin** the range of values.
- Divide the entire range of values into a series of intervals.
- Count how many values fall into each interval.

The bins are usually specified as consecutive, non-overlapping intervals of a variable.

The **matplotlib.pyplot.hist()** function plots a histogram. It computes and draws the histogram of x.



Parameters

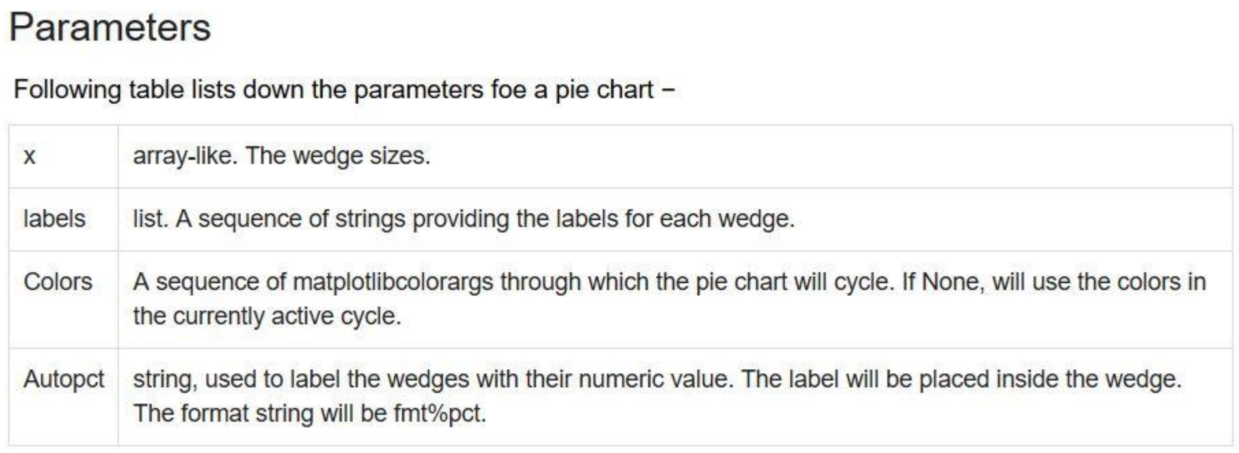
The following table lists down the parameters for a histogram −

x	array or sequence of arrays
bins	integer or sequence or 'auto', optional
optional parameters	
range	The lower and upper range of the bins.
density	If True, the first element of the return tuple will be the counts normalized to form a probability density
cumulative	If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values.
histtype	The type of histogram to draw. Default is 'bar' ▪ '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.

Pie chart

A Pie Chart can only display one series of data. Pie charts show the size of items (called wedge) in one data series, proportional to the sum of the items. The data points in a pie chart are shown as a percentage of the whole pie.

Matplotlib API has a **pie()** function that generates a pie diagram representing data in an array.



Parameters

Following table lists down the parameters foe a pie chart –

x	array-like. The wedge sizes.
labels	list. A sequence of strings providing the labels for each wedge.
Colors	A sequence of matplotlibcolorargs through which the pie chart will cycle. If None, will use the colors in the currently active cycle.
Autopct	string, used to label the wedges with their numeric value. The label will be placed inside the wedge. The format string will be fmt%pct.

Contour plot

Contour plots (sometimes called Level Plots) are a way to show a three-dimensional surface on a two-dimensional plane. It graphs two predictor variables X Y on the y-axis and a response variable Z as contours. These contours are sometimes called the z-slices or the iso-response values.

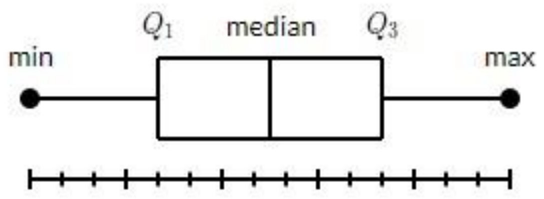
A contour plot is appropriate if you want to see how alue Z changes as a function of two inputs X and Y, such that Z = f(X,Y). A contour line or isoline of a function of two variables is a curve along which the function has a constant value.

The independent variables x and y are usually restricted to a regular grid called meshgrid. The numpy.meshgrid creates a rectangular grid out of an array of x values and an array of y values.

Matplotlib API contains contour() and contourf() functions that draw contour lines and filled contours, respectively. Both functions need three parameters x,y and z.

Box Plot

A box plot which is also known as a whisker plot displays a summary of a set of data containing the minimum, first quartile, median, third quartile, and maximum. In a box plot, we draw a box from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.



Matplotlib API contains boxplot() function that draws box plot for the data.

Violin Plot

Violin plots are similar to box plots, except that they also show the probability density of the data at different values. These plots include a marker for the median of the data and a box indicating the interquartile range, as in the standard box plots. Overlaid on this box plot is a kernel density estimation. Like box plots, violin plots are used to represent comparison of a variable distribution (or sample distribution) across different "categories".

A violin plot is more informative than a plain box plot. In fact while a box plot only shows summary statistics such as mean/median and interquartile ranges, the violin plot shows the full distribution of the data. Matplotlib API contains violinplot() function that draws violin plot for the data.

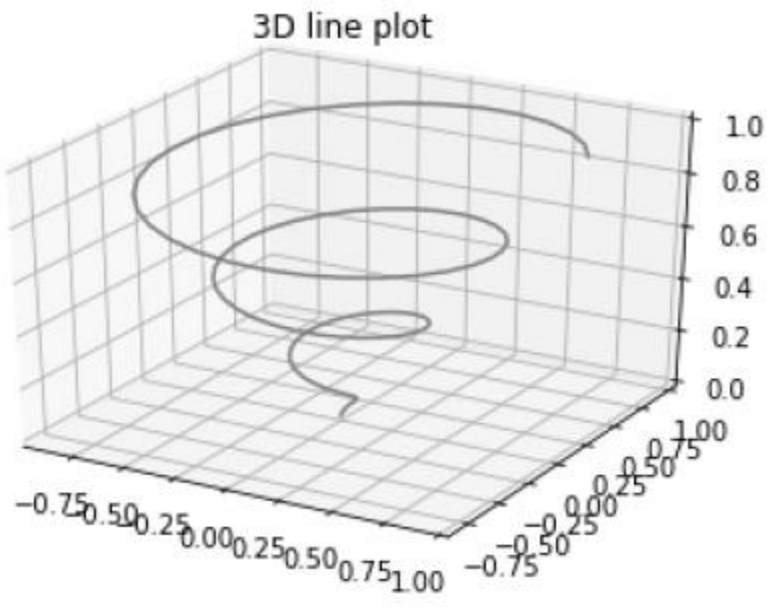
Three Dimensional plotting

Even though Matplotlib was initially designed with only two-dimensional plotting in mind, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display in later versions, to provide a set of tools for three-dimensional data visualization. Three-dimensional plots are enabled by importing the **mplot3d toolkit**, included with the Matplotlib package.

A three-dimensional axes can be created by passing the keyword projection='3d' to any of the normal axes creation routines.

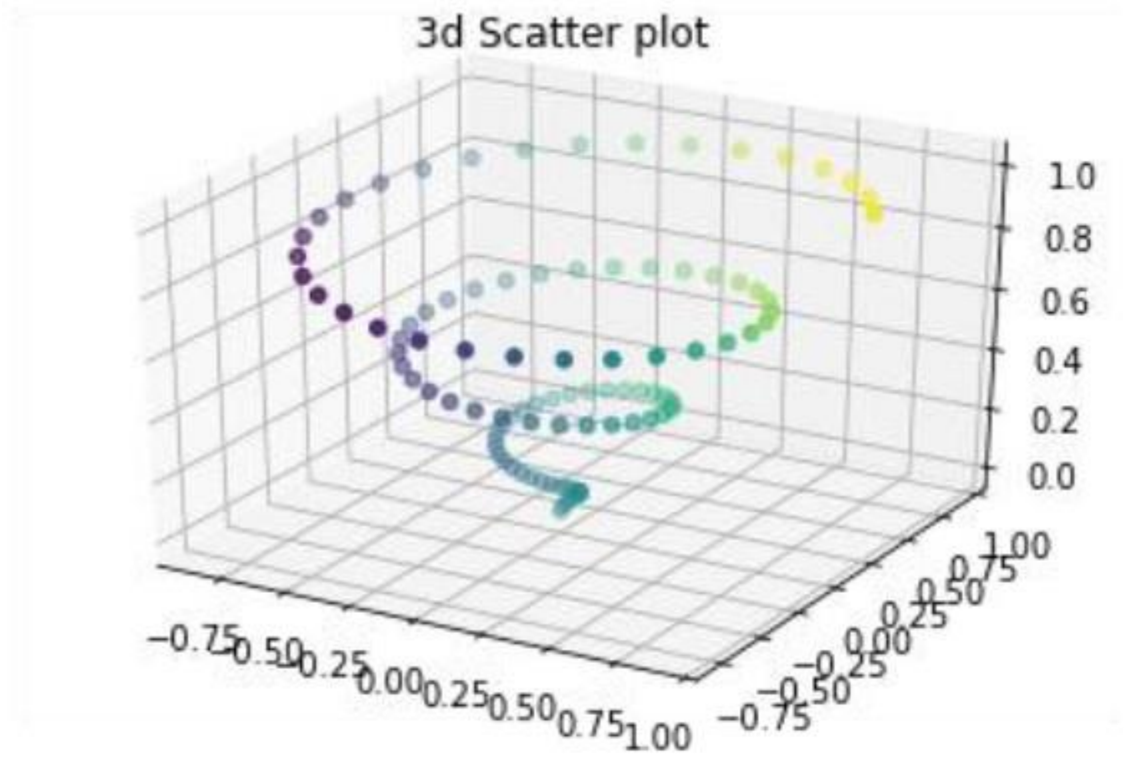
```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
z = np.linspace(0, 1, 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)
ax.plot3D(x, y, z, 'gray')
ax.set_title('3D line plot')
plt.show()
```

We can now plot a variety of three-dimensional plot types. The most basic three-dimensional plot is a **3D line plot** created from sets of (x, y, z) triples. This can be created using the ax.plot3D function.



3D scatter plot is generated by using the **ax.scatter3D** function.

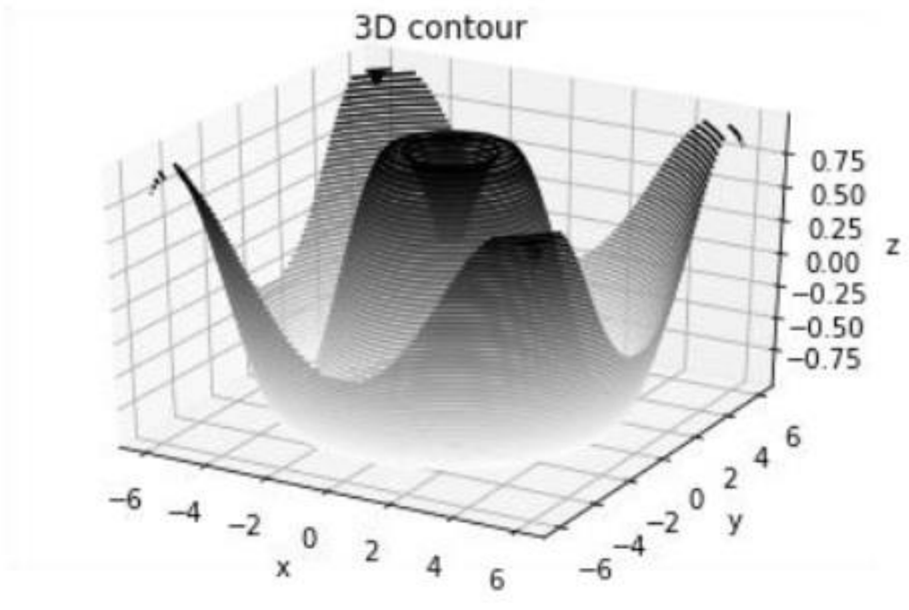
```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
z = np.linspace(0, 1, 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)
c = x + y
ax.scatter(x, y, z, c=c)
ax.set_title('3d Scatter plot')
plt.show()
```



3D CONTOUR PLOT

The **ax.contour3D()** function creates three-dimensional contour plot. It requires all the input data to be in the form of two-dimensional regular grids, with the Z-data evaluated at each point. Here, we will show a three-dimensional contour diagram of a three-dimensional sinusoidal function.

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

fig = plt.figure()
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')
ax.set_title('3D contour')
plt.show()
```

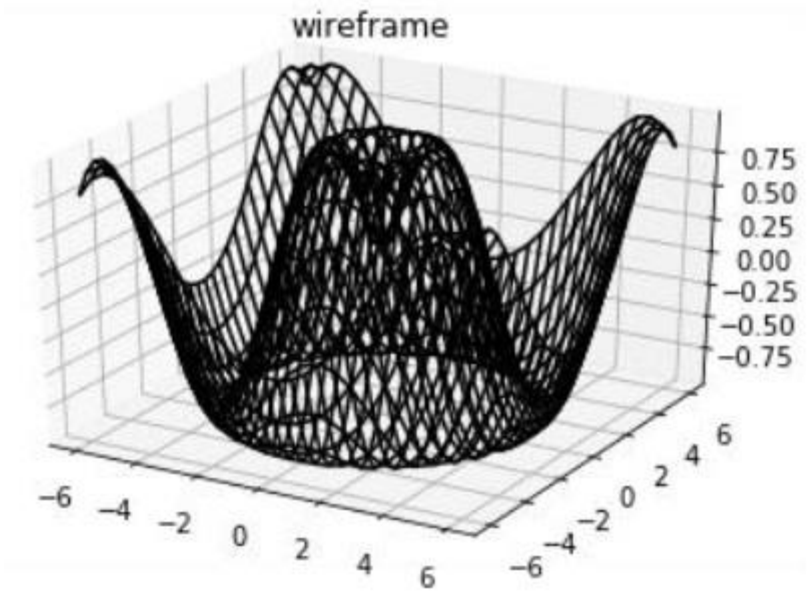


3D WIREFRAME PLOT

Wireframe plot takes a grid of values and projects it onto the specified three-dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize. The **plot_wireframe()** function is used for the purpose −

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

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe')
plt.show()
```

The above line of code will generate the following output −

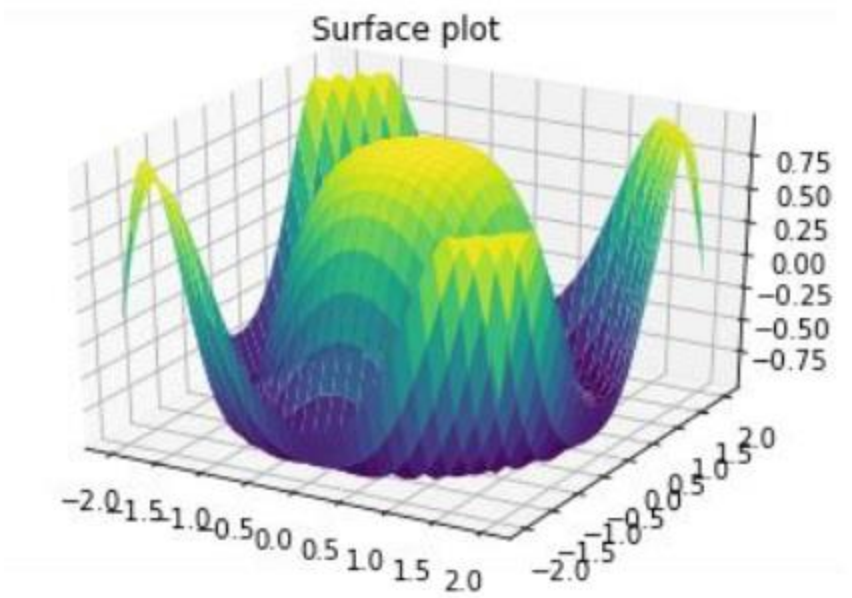


3D SURFACE PLOT

Surface plot shows a functional relationship between a designated dependent variable (Y), and two independent variables (X and Z). The plot is a companion plot to the contour plot. A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. This can aid perception of the topology of the surface being visualized. The **plot_surface()** function x,y and z as arguments.

```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
x = np.outer(np.linspace(-2, 2, 30), np.ones(30))
y = x.copy().T # transpose
z = np.cos(x ** 2 + y ** 2)

fig = plt.figure()
ax = plt.axes(projection='3d')

ax.plot_surface(x, y, z,cmap='viridis', edgecolor='none')
ax.set_title('Surface plot')
plt.show()
```

The above line of code will generate the following output −



WORKING WITH TEXT

Matplotlib has extensive text support, including support for mathematical expressions, **TrueType** support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Matplotlib includes its own matplotlib.font_manager which implements a cross platform, W3C compliant font finding algorithm.

The user has a great deal of control over text properties (font size, font weight, text location and color, etc.). Matplotlib implements a large number of TeX math symbols and commands.

The following list of commands are used to create text in the Pyplot interface −

text	Add text at an arbitrary location of the Axes.
annotate	Add an annotation, with an optional arrow, at an arbitrary location of theAxes.
xlabel	Add a label to the Axes's x-axis.
ylabel	Add a label to the Axes's y-axis.
title	Add a title to the Axes.
figtext	Add text at an arbitrary location of the Figure.
suptitle	Add a title to the Figure.

All of these functions create and return a **matplotlib.text.Text()** instance.

SEABORN

In the world of Analytics, the best way to get insights is by visualizing the data. Data can be visualized by representing it as plots which is easy to understand, explore and grasp. Such data helps in drawing the attention of key elements.

To analyse a set of data using Python, we make use of Matplotlib, a widely implemented 2D plotting library. Likewise, Seaborn is a visualization library in Python. It is built on top of Matplotlib.

Seaborn Vs Matplotlib

It is summarized that if Matplotlib "tries to make easy things easy and hard things possible", Seaborn tries to make a well-defined set of hard things easy too."

Seaborn helps resolve the two major problems faced by Matplotlib; the problems are −

- Default Matplotlib parameters
- Working with data frames

As Seaborn compliments and extends Matplotlib, the learning curve is quite gradual. If you know Matplotlib, you are already half way through Seaborn.

Important Features of Seaborn

Seaborn is built on top of Python's core visualization library Matplotlib. It is meant to serve as a complement, and not a replacement. However, Seaborn comes with some very important features. Let us see a few of them here. The features help in −

- Built in themes for styling matplotlib graphics
- Visualizing univariate and bivariate data
- Fitting in and visualizing linear regression models
- Plotting statistical time series data
- Seaborn works well with NumPy and Pandas data structures
- It comes with built in themes for styling Matplotlib graphics

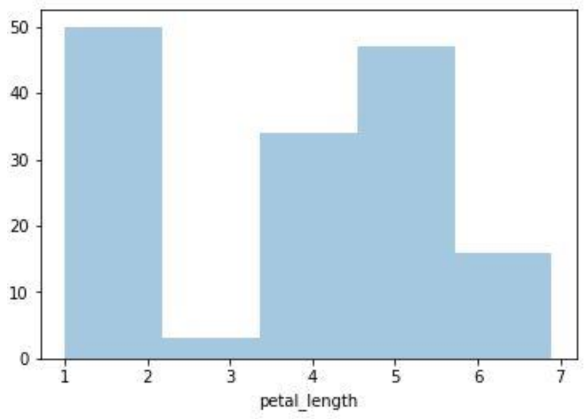
In most cases, you will still use Matplotlib for simple plotting. The knowledge of Matplotlib is recommended to tweak Seaborn's default plots.

DISTRIBUTION PLOT

It is a combination of histogram and KDE plot. Histograms represent the data distribution by forming bins along the range of the data and then drawing bars to show the number of observations that fall in each bin. Kernel Density Estimation (KDE) is a way to estimate the probability density function of a continuous random variable. It is used for non-parametric analysis.

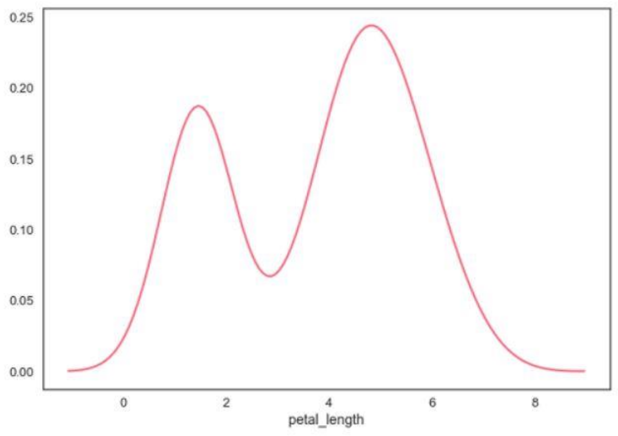
kde assigned false gives only histogram and hist assigned false gives only kde plot.

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.distplot(df['petal_length'],kde = False)
plt.show()
```



```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.distplot(df['petal_length'],hist=False)
plt.show()
```

Output



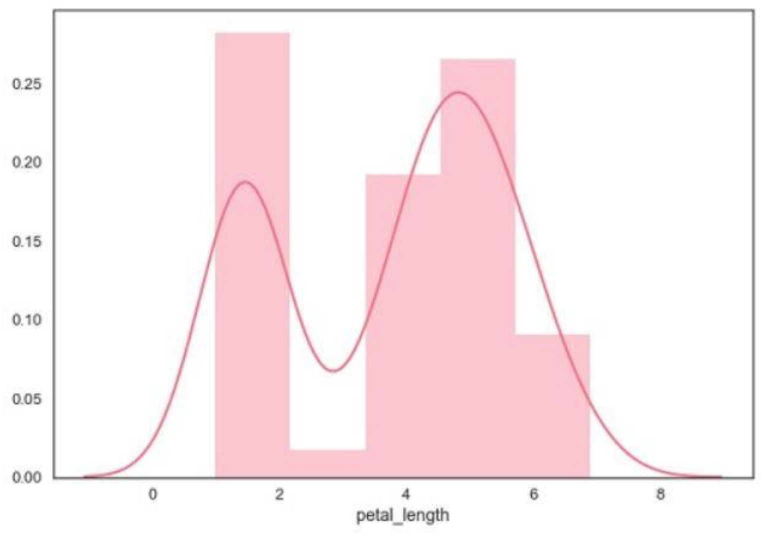
Fitting Parametric Distribution

distplot() is used to visualize the parametric distribution of a dataset.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.distplot(df['petal_length'])
plt.show()
```

Output



Plotting Bivariate Distribution

Bivariate Distribution is used to determine the relation between two variables. This mainly deals with relationship between two variables and how one variable is behaving with respect to the other.

The best way to analyze Bivariate Distribution in seaborn is by using the **jointplot()** function.

Jointplot creates a multi-panel figure that projects the bivariate relationship between two variables and also the univariate distribution of each variable on separate axes.

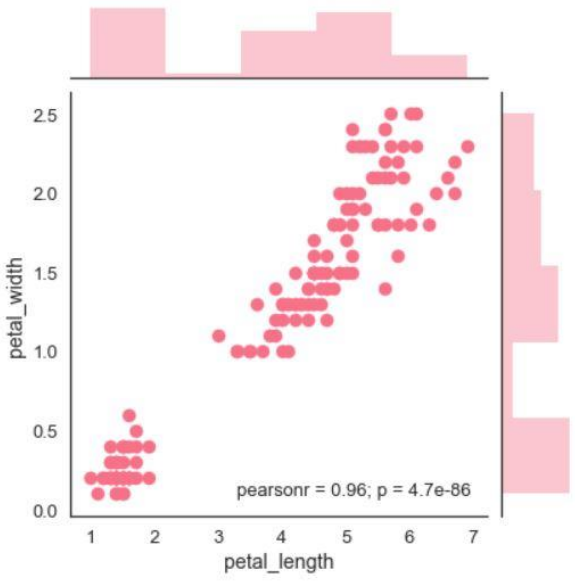
Scatter Plot

Scatter plot is the most convenient way to visualize the distribution where each observation is represented in two-dimensional plot via x and y axis.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.jointplot(x = 'petal_length',y = 'petal_width',data = df)
plt.show()
```

Output



The above figure shows the relationship between the **petal_length** and **petal_width** in the Iris data. A trend in the plot says that positive correlation exists between the variables under study.

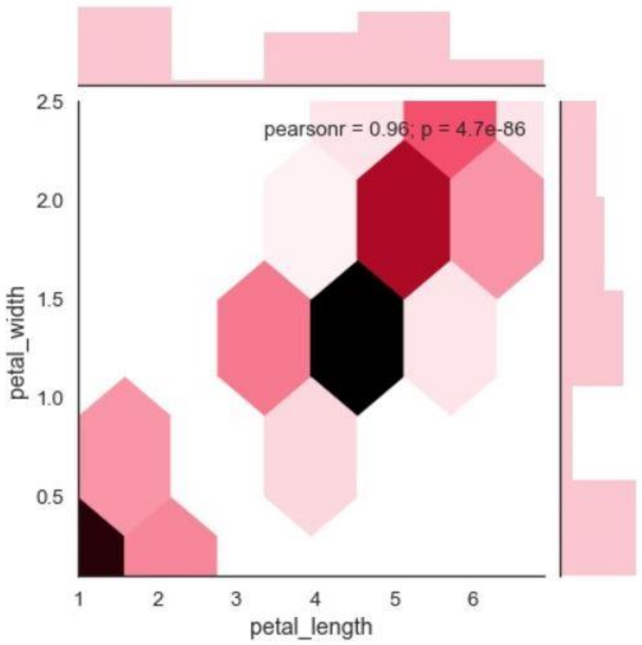
Hexbin Plot

Hexagonal binning is used in bivariate data analysis when the data is sparse in density i.e., when the data is very scattered and difficult to analyze through scatterplots.

An addition parameter called ‘kind’ and value ‘hex’ plots the hexbin plot.

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
```

```
df = sb.load_dataset('iris')
sb.jointplot(x = 'petal_length',y = 'petal_width',data = df,kind = 'hex')
plt.show()
```



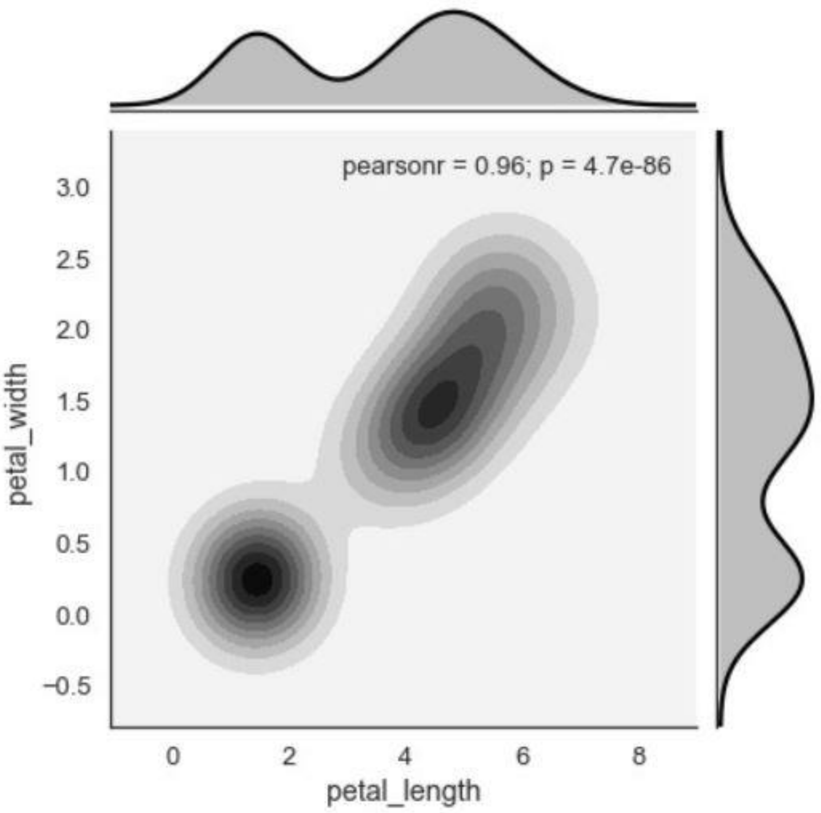
Kernel Density Estimation

Kernel density estimation is a non-parametric way to estimate the distribution of a variable. In seaborn, we can plot a kde using **jointplot().**

Pass value ‘kde’ to the parameter kind to plot kernel plot.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.jointplot(x = 'petal_length',y = 'petal_width',data = df,kind = 'hex')
plt.show()
```



VISUALIZING PAIRWISE RELATIONSHIP

Datasets under real-time study contain many variables. In such cases, the relation between each and every variable should be analyzed. Plotting Bivariate Distribution for (n,2) combinations will be a very complex and time taking process.

To plot multiple pairwise bivariate distributions in a dataset, you can use the **pairplot()** function. This shows the relationship for (n,2) combination of variable in a DataFrame as a matrix of plots and the diagonal plots are the univariate plots.

Axes

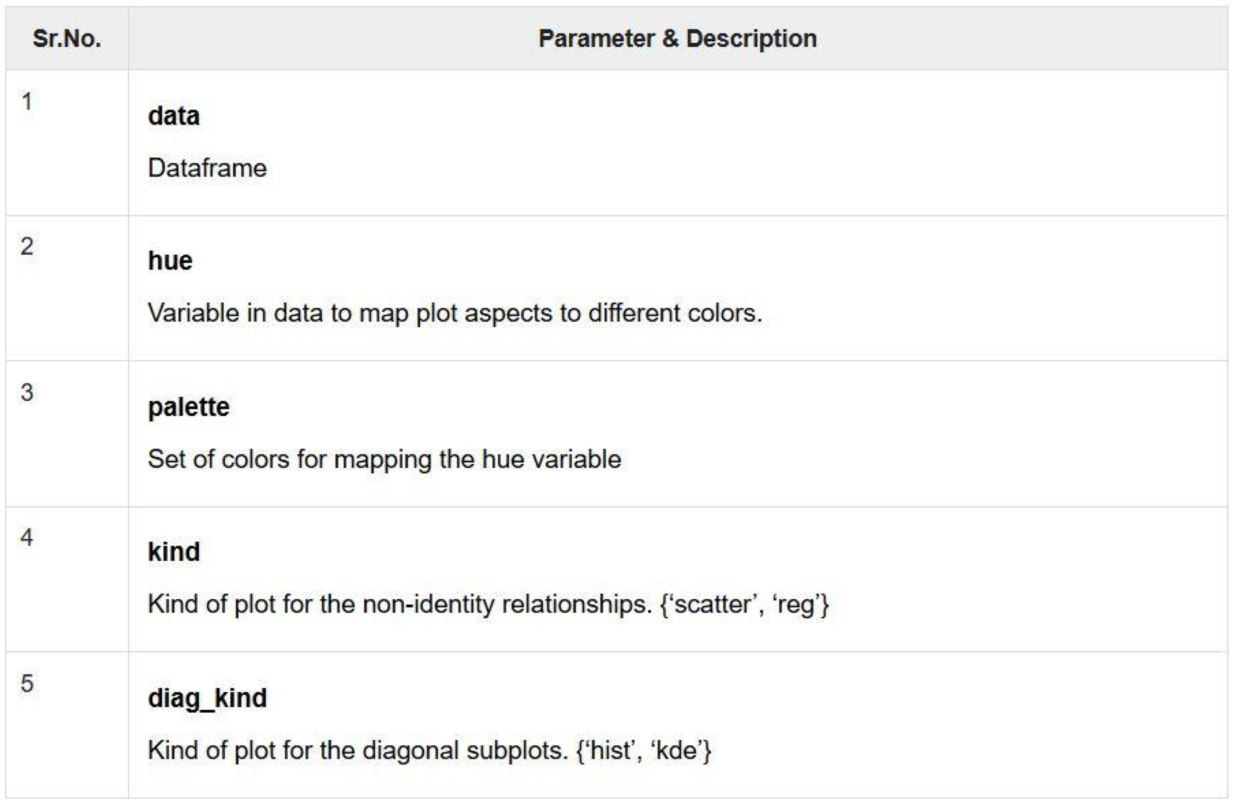
In this section, we will learn what are Axes, their usage, parameters, and so on.

Usage

```
seaborn.pairplot(data,…)
```

Parameters

Following table lists down the parameters for Axes −



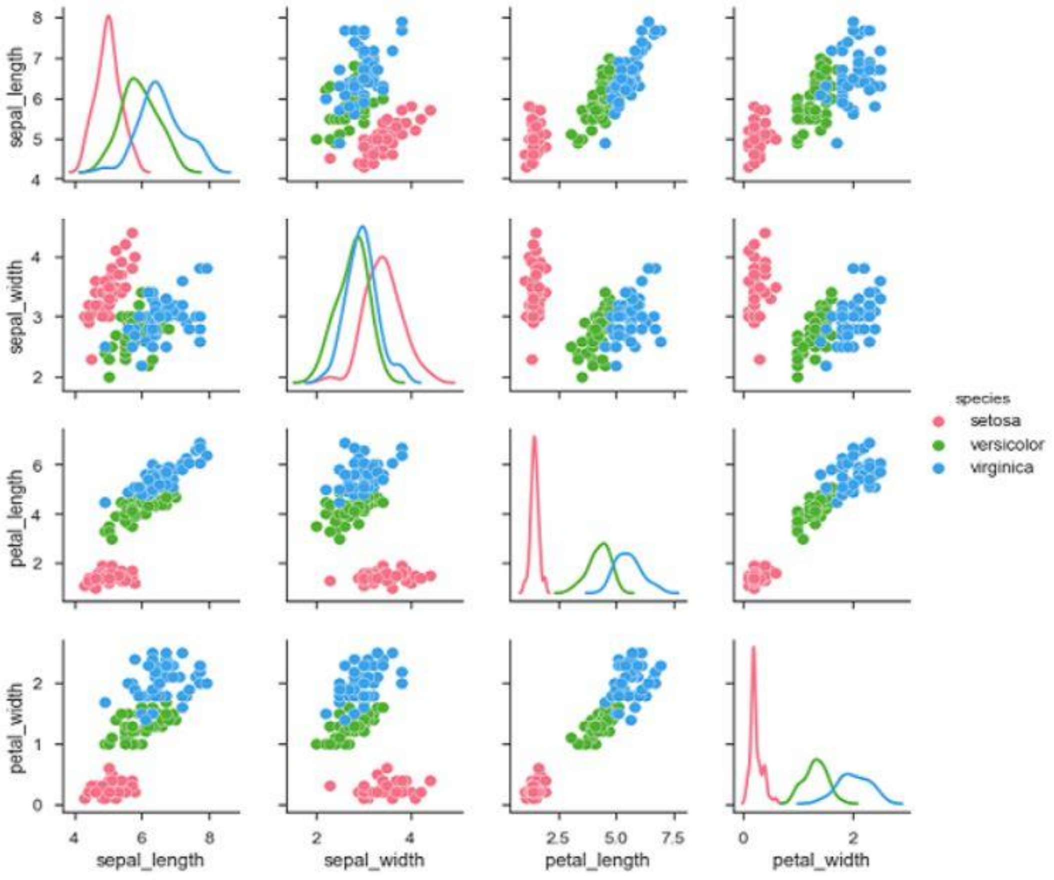
Sr.No.	Parameter & Description
1	**data** Dataframe
2	**hue** Variable in data to map plot aspects to different colors.
3	**palette** Set of colors for mapping the hue variable
4	**kind** Kind of plot for the non-identity relationships. {‘scatter’, ‘reg’}
5	**diag_kind** Kind of plot for the diagonal subplots. {‘hist’, ‘kde’}

Except data, all other parameters are optional. There are few other parameters which **pairplot** can accept. The above mentioned are often used params.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.set_style("ticks")
sb.pairplot(df,hue = 'species',diag_kind = "kde",kind = "scatter",palette =
"husl")
plt.show()
```

Output



We can observe the variations in each plot. The plots are in matrix format where the row name represents x axis and column name represents the y axis.

Categorical Scatter Plots

In this section, we will learn about categorical scatter plots.

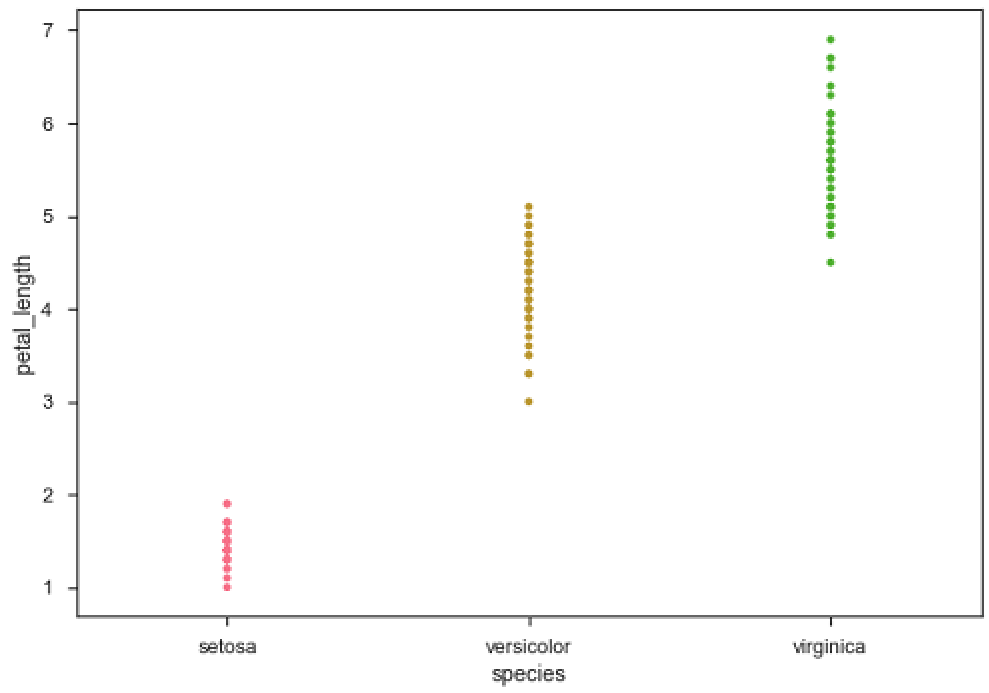
stripplot()

stripplot() is used when one of the variable under study is categorical. It represents the data in sorted order along any one of the axis.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.stripplot(x = "species", y = "petal_length", data = df)
plt.show()
```

Output



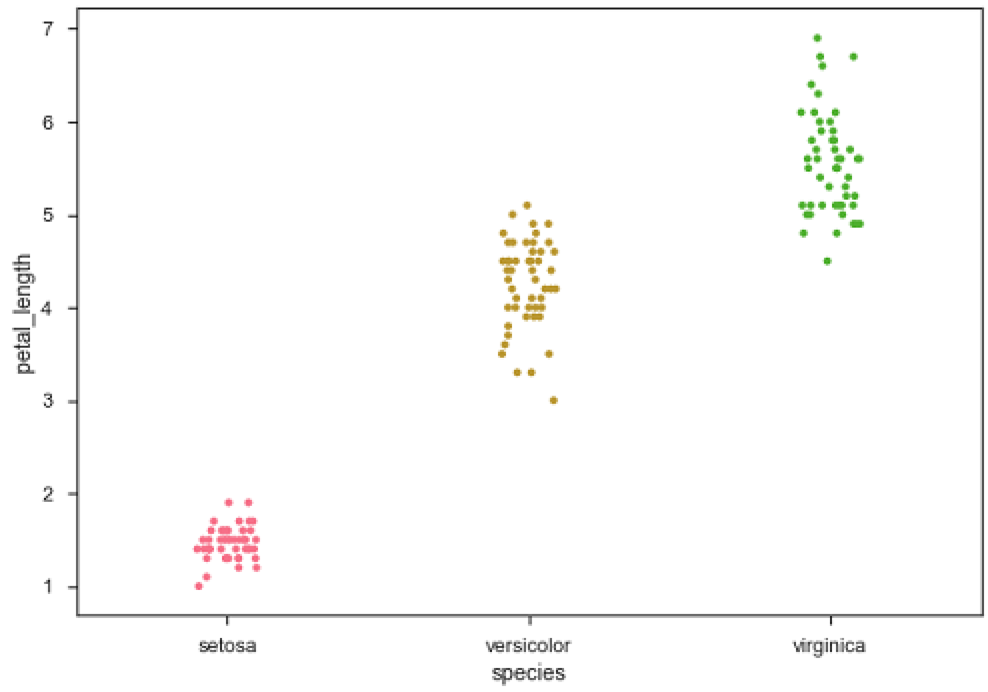
In the above plot, we can clearly see the difference of **petal_length** in each species. But, the major problem with the above scatter plot is that the points on the scatter plot are overlapped. We use the ‘Jitter’ parameter to handle this kind of scenario.

Jitter adds some random noise to the data. This parameter will adjust the positions along the categorical axis.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.stripplot(x = "species", y = "petal_length", data = df, jitter = Ture)
plt.show()
```

Output



Now, the distribution of points can be seen easily.

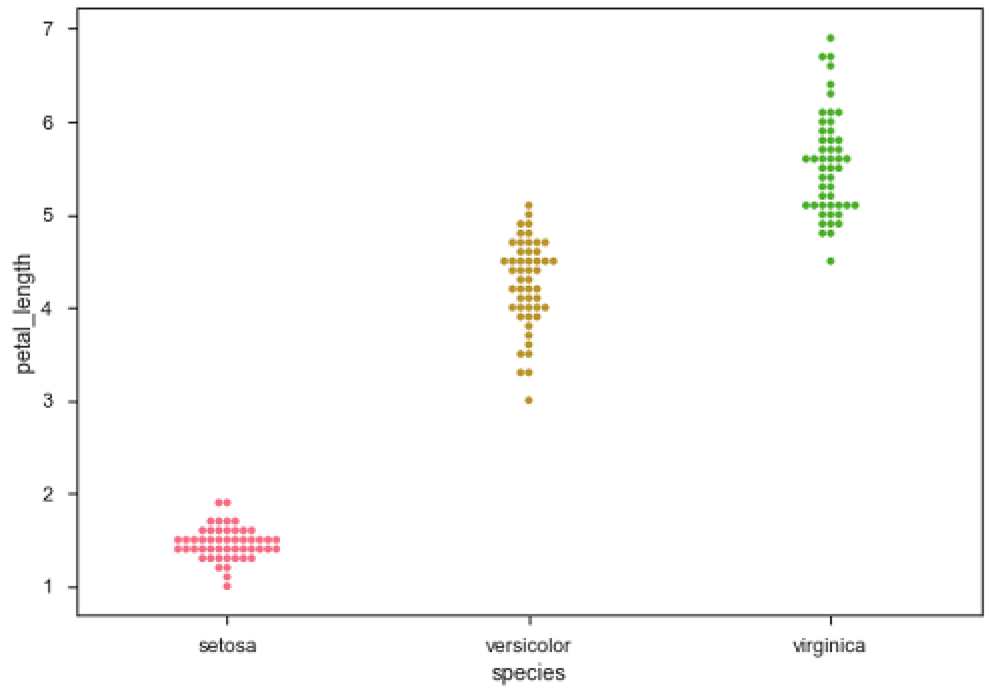
Swarmplot()

Another option which can be used as an alternate to ‘Jitter’ is function **swarmplot()**. This function positions each point of scatter plot on the categorical axis and thereby avoids overlapping points −

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.swarmplot(x = "species", y = "petal_length", data = df)
plt.show()
```

Output



Box Plots

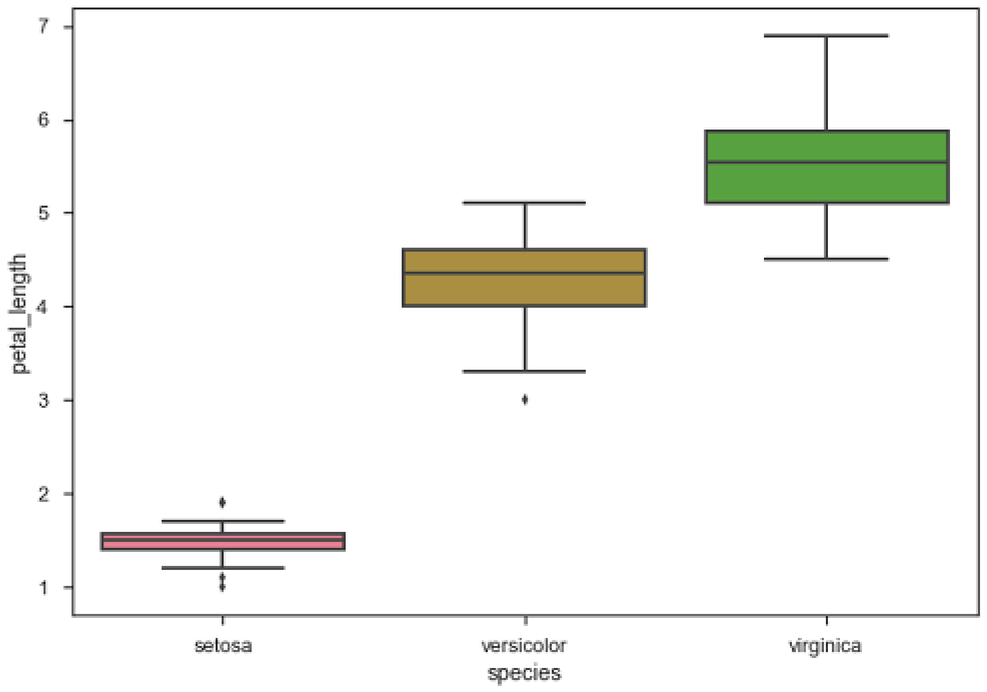
Boxplot is a convenient way to visualize the distribution of data through their quartiles.

Box plots usually have vertical lines extending from the boxes which are termed as whiskers. These whiskers indicate variability outside the upper and lower quartiles, hence Box Plots are also termed as **box-and-whisker** plot and **box-and-whisker** diagram. Any Outliers in the data are plotted as individual points.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.swarmplot(x = "species", y = "petal_length", data = df)
plt.show()
```

Output



The dots on the plot indicates the outlier.

Violin Plots

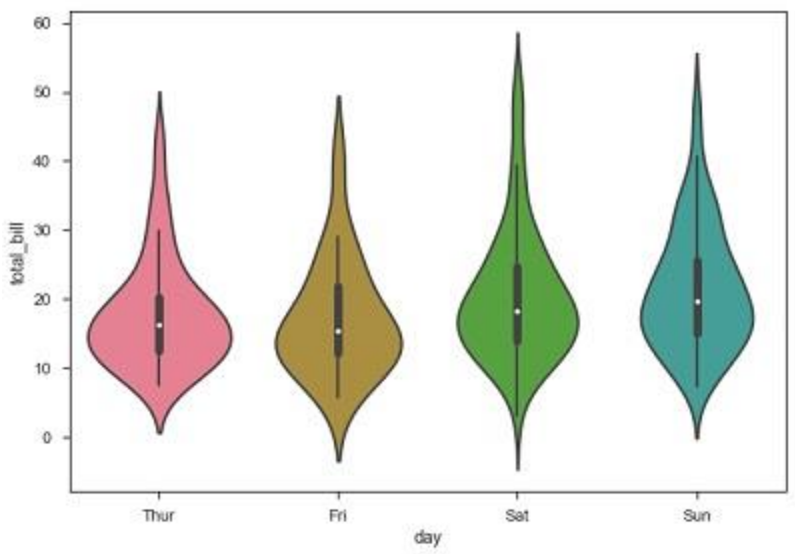
Violin Plots are a combination of the box plot with the kernel density estimates. So, these plots are easier to analyze and understand the distribution of the data.

Let us use tips dataset called to learn more into violin plots. This dataset contains the information related to the tips given by the customers in a restaurant.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('tips')
sb.violinplot(x = "day", y = "total_bill", data=df)
plt.show()
```

Output



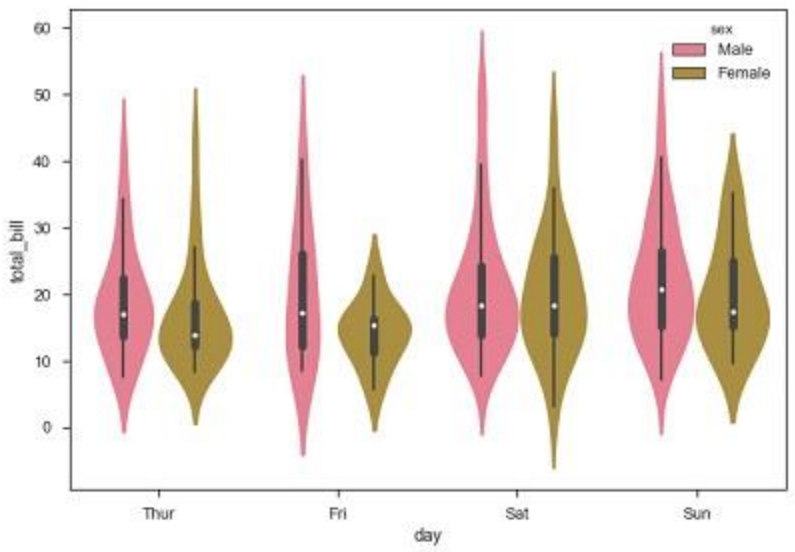
The quartile and whisker values from the boxplot are shown inside the violin. As the violin plot uses KDE, the wider portion of violin indicates the higher density and narrow region represents relatively lower density. The Inter-Quartile range in boxplot and higher density portion in kde fall in the same region of each category of violin plot.

The above plot shows the distribution of total_bill on four days of the week. But, in addition to that, if we want to see how the distribution behaves with respect to sex, lets explore it in below example.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('tips')
sb.violinplot(x = "day", y = "total_bill",hue = 'sex', data = df)
plt.show()
```

Output



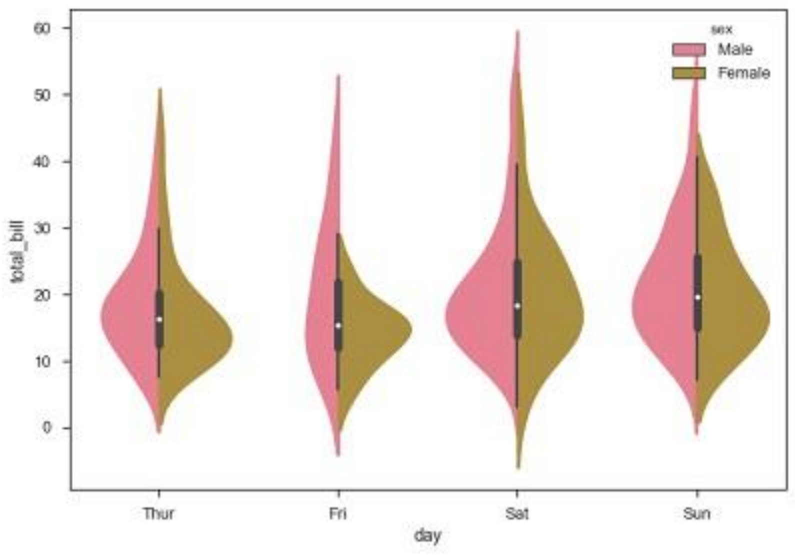
Now we can clearly see the spending behavior between male and female. We can easily say that, men make more bill than women by looking at the plot.

And, if the hue variable has only two classes, we can beautify the plot by splitting each violin into two instead of two violins on a given day. Either parts of the violin refer to each class in the hue variable.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('tips')
sb.violinplot(x = "day", y="total_bill",hue = 'sex', data = df)
plt.show()
```

Output



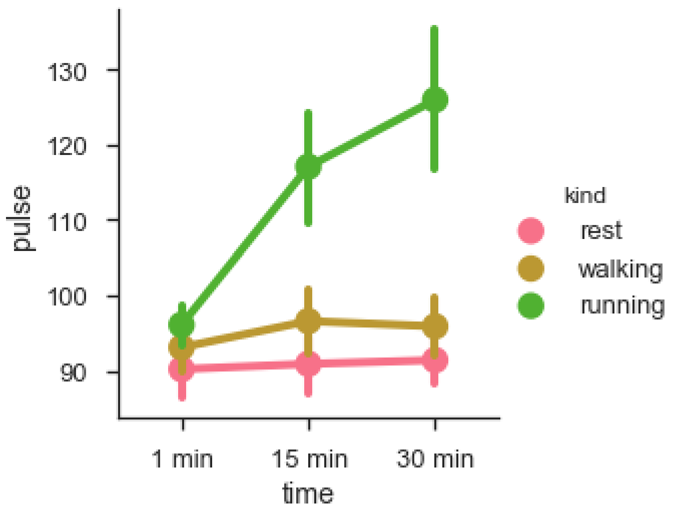
Factorplot

Factorplot draws a categorical plot on a FacetGrid. Using ‘kind’ parameter we can choose the plot like boxplot, violinplot, barplot and stripplot. FacetGrid uses pointplot by default.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('exercise')
sb.factorplot(x = "time", y = pulse", hue = "kind",data = df);
plt.show()
```

Output

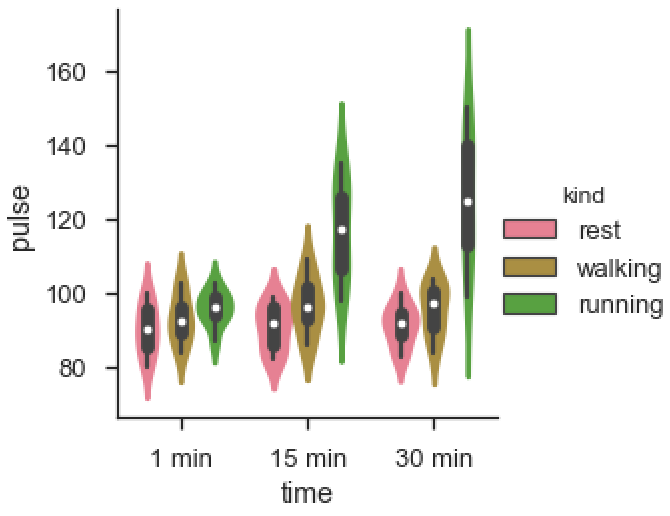


We can use different plot to visualize the same data using the **kind** parameter.

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('exercise')
sb.factorplot(x = "time", y = "pulse", hue = "kind", kind = 'violin',data =
df);
plt.show()
```

Output



In factorplot, the data is plotted on a facet grid.

What is Facet Grid?

Facet grid forms a matrix of panels defined by row and column by dividing the variables. Due of panels, a single plot looks like multiple plots. It is very helpful to analyze all combinations in two discrete variables.

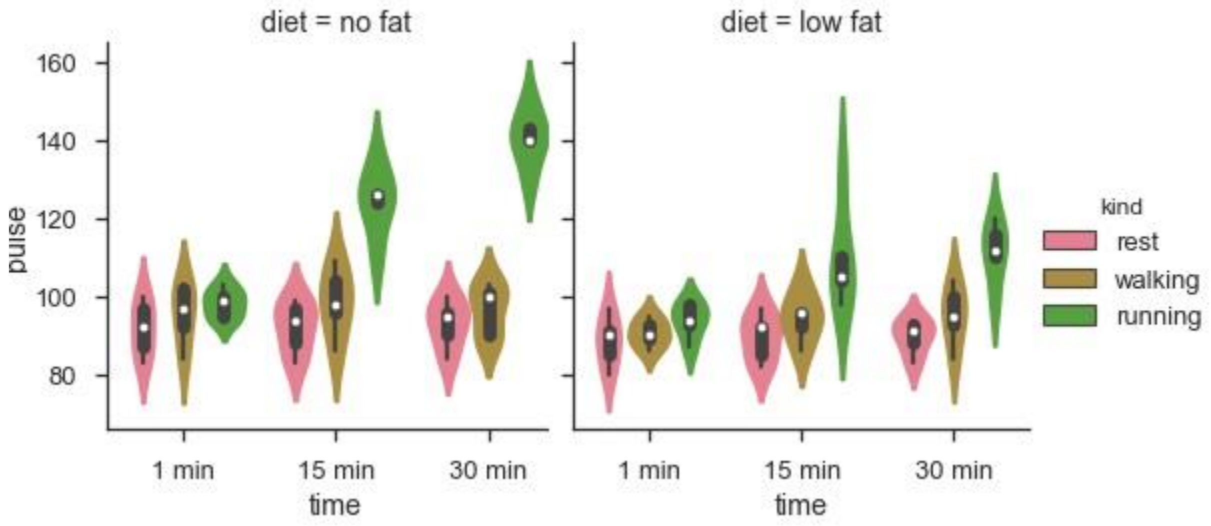
Let us visualize the above the definition with an example

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('exercise')
sb.factorplot(x = "time", y = "pulse", hue = "kind", kind = 'violin', col =
"diet", data = df);
```

```
plt.show()
```

Output



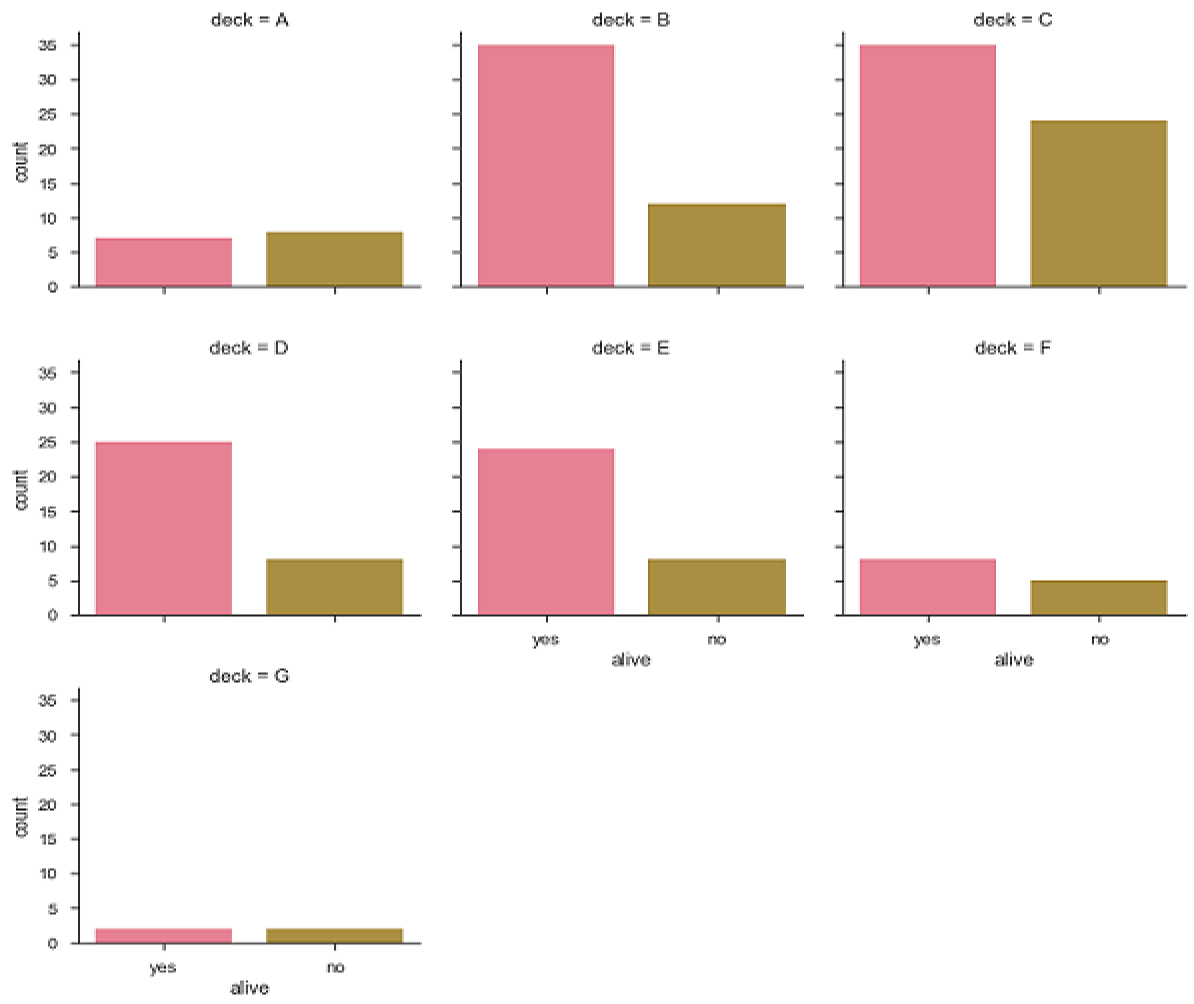
The advantage of using Facet is, we can input another variable into the plot. The above plot is divided into two plots based on a third variable called ‘diet’ using the ‘col’ parameter.

We can make many column facets and align them with the rows of the grid −

Example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('titanic')
sb.factorplot("alive", col = "deck", col_wrap = 3,data =
df[df.deck.notnull()],kind = "count")
plt.show()
```

output



SEABORN-LINEAR RELATIONSHIP

Functions to Draw Linear Regression Models

There are main functions in Seaborn to visualize a linear relationship determined through regression is regplot().

regplot

accepts the x and y variables in a variety of formats including simple numpy arrays, pandas Series objects, or as references to variables in a pandas DataFrame

Let us now draw the plots.

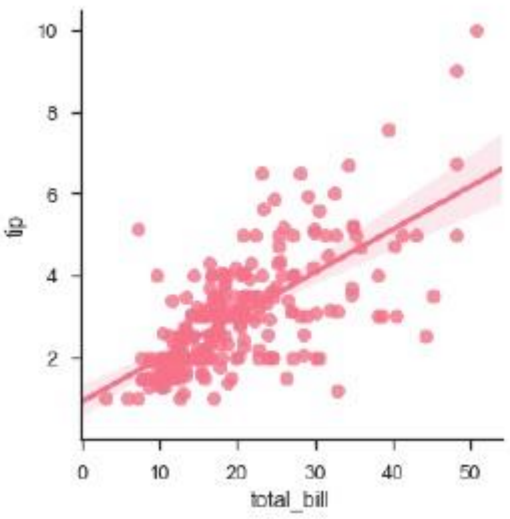
Example

Plotting the regplot and then lmplot with the same data in this example

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('tips')
sb.regplot(x = "total_bill", y = "tip", data = df)
plt.show()
```

Output

You can see the difference in the size between two plots.



We can also fit a linear regression when one of the variables takes discrete values

SCIPY

SciPy, pronounced as Sigh Pi, is a scientific python open source, distributed under the BSD licensed library to perform Mathematical, Scientific and Engineering Computations.

The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The SciPy library is built to work with NumPy arrays and provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization. Together, they run on all popular operating systems, are quick to install and are free of charge. NumPy and SciPy are easy to use, but powerful enough to depend on by some of the world's leading scientists and engineers.

SciPy Sub-packages

SciPy is organized into sub-packages covering different scientific computing domains. These are summarized in the following table −

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions
scipy.stats	Statistics

Data Structure

The basic data structure used by SciPy is a multidimensional array provided by the NumPy module. NumPy provides some functions for Linear Algebra, Fourier Transforms and Random Number Generation, but not with the generality of the equivalent functions in SciPy.

SCIPY-CONSTANT PACKAGE

The **scipy.constants package** provides various constants. We have to import the required constant and use them as per the requirement. Let us see how these constant variables are imported and used.

To start with, let us compare the ‘pi’ value by considering the following example.

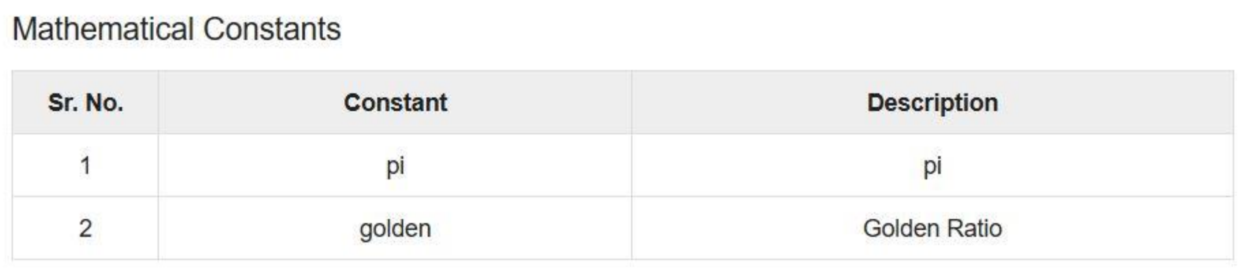
```
#Import pi constant from both the packages
from scipy.constants import pi
from math import pi

print("sciPy - pi = %.16f"%scipy.constants.pi)
print("math - pi = %.16f"%math.pi)
```

The above program will generate the following output.

```
sciPy - pi = 3.1415926535897931
math - pi = 3.1415926535897931
```

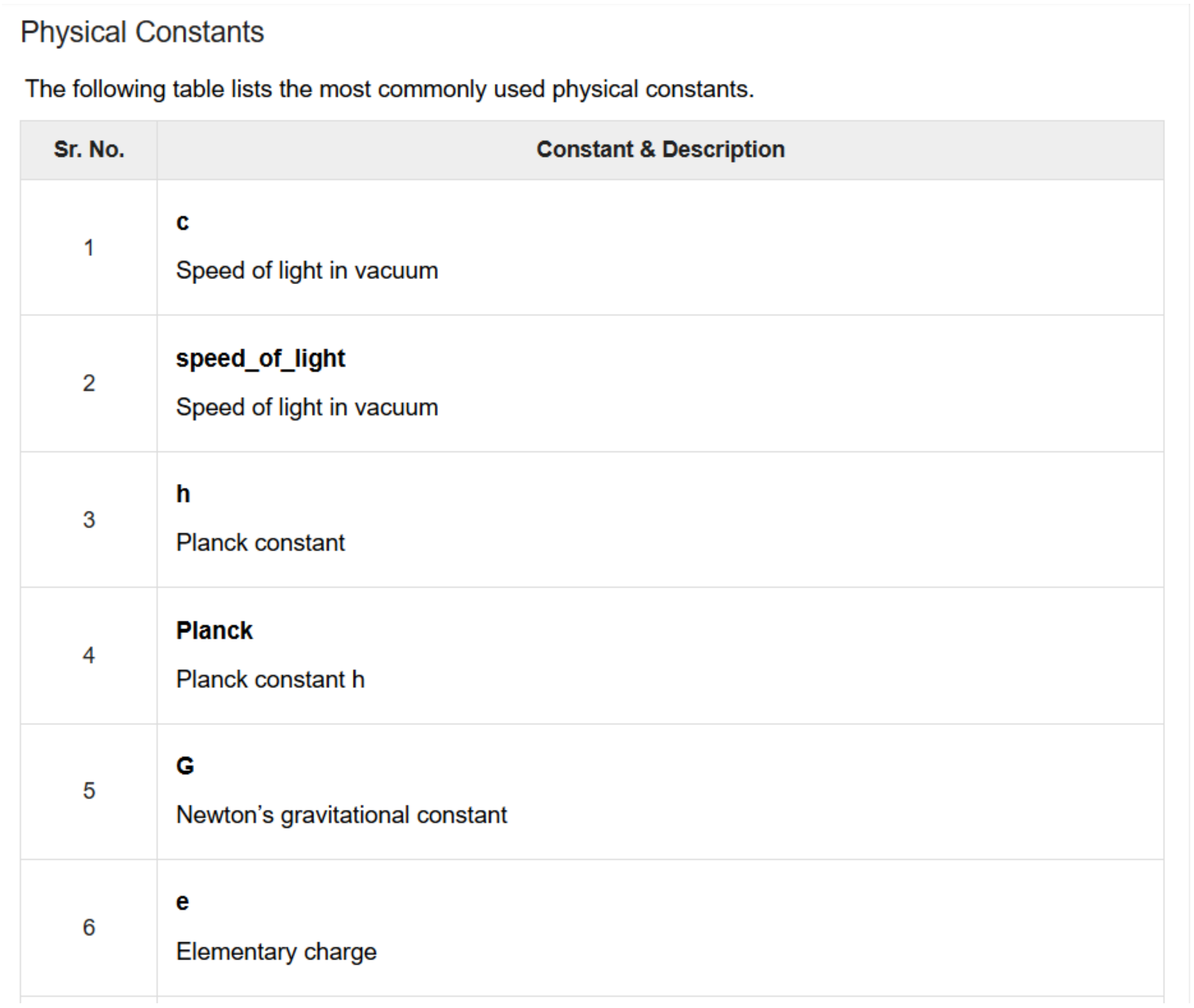
List of Constants Available

The following tables describe in brief the various constants.



Mathematical Constants

Sr. No.	Constant	Description
1	pi	pi
2	golden	Golden Ratio



Physical Constants

The following table lists the most commonly used physical constants.

Sr. No.	Constant & Description
1	**c** Speed of light in vacuum
2	**speed_of_light** Speed of light in vacuum
3	**h** Planck constant
4	**Planck** Planck constant h
5	**G** Newton’s gravitational constant
6	**e** Elementary charge

FFT PACK

Fourier Transformation is computed on a time domain signal to check its behavior in the frequency domain. Fourier transformation finds its application in disciplines such as signal and noise processing, image processing, audio signal processing, etc. SciPy offers the fftpack module, which lets the user compute fast Fourier transforms.

Following is an example of a sine function, which will be used to calculate Fourier transform using the fftpack module.

Fast Fourier Transform

Let us understand what fast Fourier transform is in detail.

One Dimensional Discrete Fourier Transform

The FFT y[k] of length N of the length-N sequence x[n] is calculated by fft() and the inverse transform is calculated using ifft(). Let us consider the following example

```
#Importing the fft and inverse fft functions from fftpackage
from scipy.fftpack import fft

#create an array with random n numbers
x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])

#Applying the fft function
y = fft(x)
print y
```

The above program will generate the following output.

```
[ 4.50000000+0.j           2.08155948-1.65109876j   -1.83155948+1.60822041j
 -1.83155948-1.60822041j   2.08155948+1.65109876j ]
```

Let us look at another example

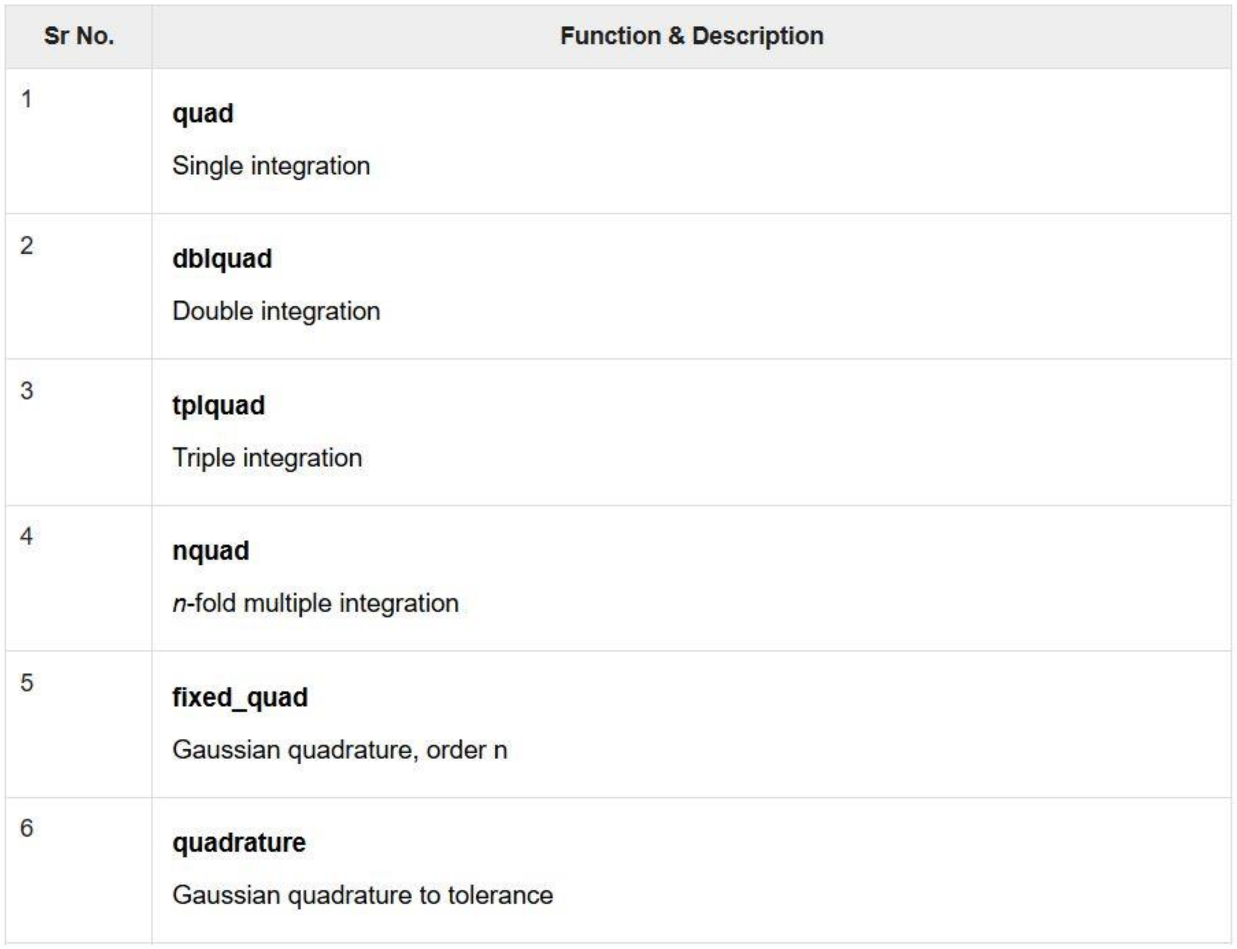
```
#FFT is already in the workspace, using the same workspace to for inverse
transform

yinv = ifft(y)

print yinv
```

The above program will generate the following output.

```
[ 1.0+0.j   2.0+0.j   1.0+0.j   -1.0+0.j   1.5+0.j ]
```

SciPy-Integrate

When a function cannot be integrated analytically, or is very difficult to integrate analytically, one generally turns to numerical integration methods. SciPy has a number of routines for performing numerical integration. Most of them are found in the same **scipy.integrate** library. The following table lists some commonly used functions.



Sr No.	Function & Description
1	**quad** Single integration
2	**dblquad** Double integration
3	**tplquad** Triple integration
4	**nquad** *n*-fold multiple integration
5	**fixed_quad** Gaussian quadrature, order n
6	**quadrature** Gaussian quadrature to tolerance

Single Integrals

The Quad function is the workhorse of SciPy's integration functions. Numerical integration is sometimes called **quadrature**, hence the name. It is normally the default choice for performing single integrals of a function *f(x)* over a given fixed range from a to b.

$$\int_{a}^{b} f(x)dx$$

The general form of quad is **scipy.integrate.quad(f, a, b)**, Where 'f' is the name of the function to be integrated. Whereas, 'a' and 'b' are the lower and upper limits, respectively. Let us see an example of the Gaussian function, integrated over a range of 0 and 1.

We first need to define the function → $f(x) = e^{-x^2}$, this can be done using a lambda expression and then call the quad method on that function.

```
import scipy.integrate
from numpy import exp
f= lambda x:exp(-x**2)
i = scipy.integrate.quad(f, 0, 1)
print i
```

The above program will generate the following output.

```
(0.7468241328124271, 8.291413475940725e-15)
```

The quad function returns the two values, in which the first number is the value of integral and the second value is the estimate of the absolute error in the value of integral.

Note − Since quad requires the function as the first argument, we cannot directly pass exp as the argument. The Quad function accepts positive and negative infinity as limits. The Quad function can integrate standard predefined NumPy functions of a single variable, such as exp, sin and cos.

Multiple Integrals

The mechanics for double and triple integration have been wrapped up into the functions **dblquad, tplquad** and **nquad**. These functions integrate four or six arguments, respectively. The limits of all inner integrals need to be defined as functions.

Double Integrals

The general form of **dblquad** is scipy.integrate.dblquad(func, a, b, gfun, hfun). Where, func is the name of the function to be integrated, ‘a’ and ‘b’ are the lower and upper limits of the x variable, respectively, while gfun and hfun are the names of the functions that define the lower and upper limits of the y variable.

As an example, let us perform the double integral method.

$$\int_{0}^{1/2} dy \int_{0}^{\sqrt{1-4y^2}} 16xy \:dx$$

We define the functions f, g, and h, using the lambda expressions. Note that even if g and h are constants, as they may be in many cases, they must be defined as functions, as we have done here for the lower limit.

```
import scipy.integrate
from numpy import exp
from math import sqrt
f = lambda x, y : 16*x*y
g = lambda x : 0
h = lambda y : sqrt(1-4*y**2)
i = scipy.integrate.dblquad(f, 0, 0.5, g, h)
print i
```

The above program will generate the following output.

```
(0.5, 1.7092350012594845e-14)
```

SciPy-Interpolate

What is Interpolation?

Interpolation is the process of finding a value between two points on a line or a curve. To help us remember what it means, we should think of the first part of the word, 'inter,' as meaning 'enter,' which reminds us to look 'inside' the data we originally had. This tool, interpolation, is not only useful in statistics, but is also useful in science, business, or when there is a need to predict values that fall within two existing data points.

Let us create some data and see how this interpolation can be done using the **scipy.interpolate** package.

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt
x = np.linspace(0, 4, 12)
y = np.cos(x**2/3+4)
print x,y
```

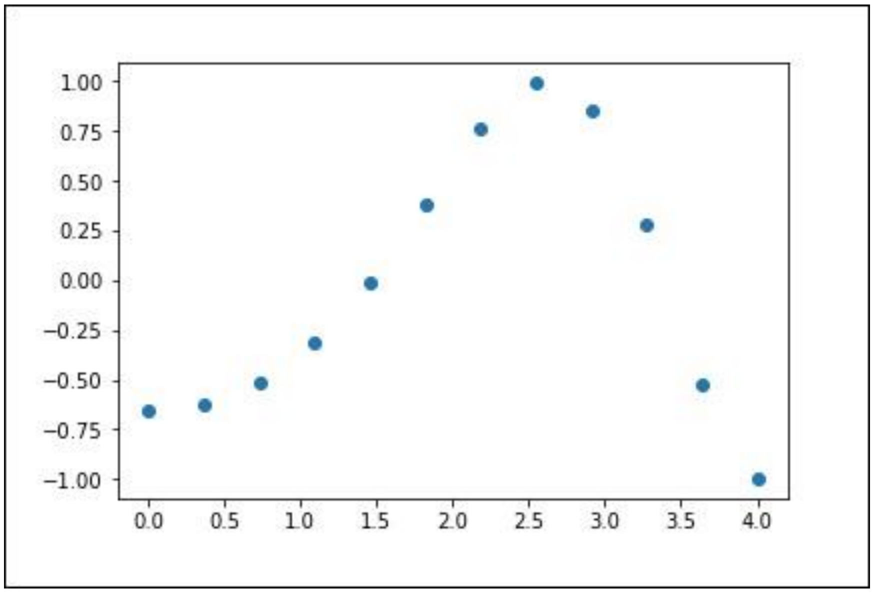
The above program will generate the following output.

```
(
   array([0.,  0.36363636,  0.72727273,  1.09090909,  1.45454545, 1.81818182,
          2.18181818,  2.54545455,  2.90909091,  3.27272727,  3.63636364,
4.]),

   array([-0.65364362,  -0.61966189,  -0.51077021,  -0.31047698,  -
0.00715476,
           0.37976236,   0.76715099,   0.99239518,   0.85886263,
0.27994201,
          -0.52586509,  -0.99582185])
)
```

Now, we have two arrays. Assuming those two arrays as the two dimensions of the points in space, let us plot using the following program and see how they look like.

```
plt.plot(x, y,'o')
plt.show()
```

The above program will generate the following output.



1-D Interpolation

The interp1d class in the scipy.interpolate is a convenient method to create a function based on fixed data points, which can be evaluated anywhere within the domain defined by the given data using linear interpolation.

By using the above data, let us create a interpolate function and draw a new interpolated graph.

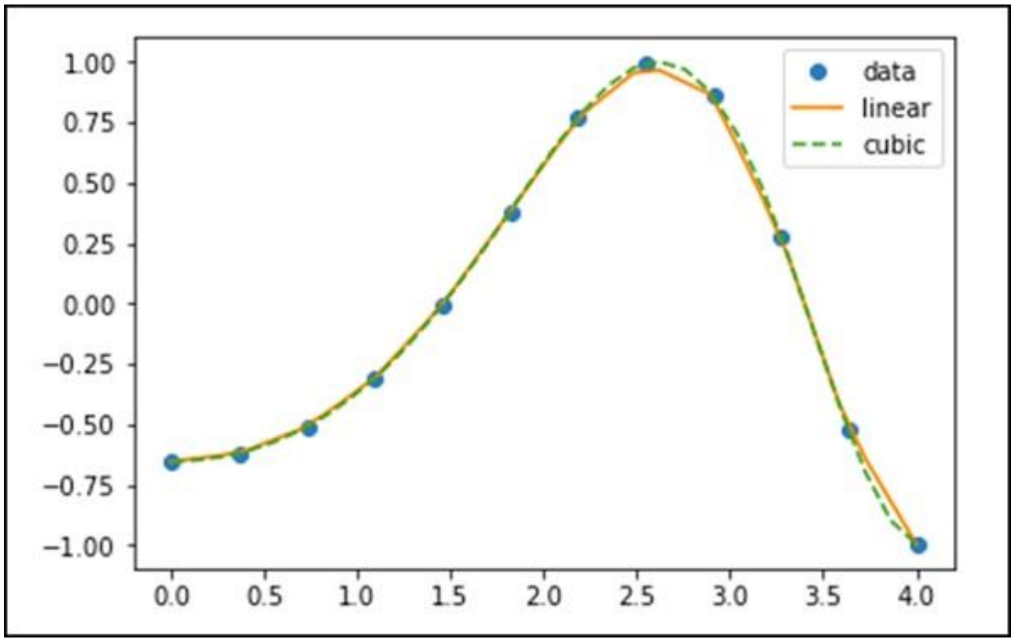
```
f1 = interp1d(x, y,kind = 'linear')

f2 = interp1d(x, y, kind = 'cubic')
```

Using the interp1d function, we created two functions f1 and f2. These functions, for a given input x returns y. The third variable kind represents the type of the interpolation technique. 'Linear', 'Nearest', 'Zero', 'Slinear', 'Quadratic', 'Cubic' are a few techniques of interpolation.

Now, let us create a new input of more length to see the clear difference of interpolation. We will use the same function of the old data on the new data.

```
xnew = np.linspace(0, 4,30)

plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')

plt.legend(['data', 'linear', 'cubic','nearest'], loc = 'best')

plt.show()
```

The above program will generate the following output.



SciPy-Linalg

SciPy is built using the optimized **ATLAS LAPACK** and **BLAS** libraries. It has very fast linear algebra capabilities. All of these linear algebra routines expect an object that can be converted into a two-dimensional array. The output of these routines is also a two-dimensional array.

SciPy.linalg vs NumPy.linalg

A scipy.linalg contains all the functions that are in numpy.linalg. Additionally, scipy.linalg also has some other advanced functions that are not in numpy.linalg. Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed.

Linear Equations

The **scipy.linalg.solve** feature solves the linear equation a * x + b * y = Z, for the unknown x, y values.

As an example, assume that it is desired to solve the following simultaneous equations.

$$x + 3y + 5z = 10$$

$$2x + 5y + z = 8$$

$$2x + 3y + 8z = 3$$

To solve the above equation for the x, y, z values, we can find the solution vector using a matrix inverse as shown below.

$$\begin{bmatrix} x\\ y\\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5\\ 2 & 5 & 1\\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10\\ 8\\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232\\ 129\\ 19 \end{bmatrix} = \begin{bmatrix} -9.28\\ 5.16\\ 0.76 \end{bmatrix}.$$

However, it is better to use the **linalg.solve** command, which can be faster and more numerically stable.

The solve function takes two inputs ‘a’ and ‘b’ in which ‘a’ represents the coefficients and ‘b’ represents the respective right hand side value and returns the solution array.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy arrays
a = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
b = np.array([2, 4, -1])

#Passing the values to the solve function
x = linalg.solve(a, b)

#printing the result array
print x
```

The above program will generate the following output.

```
array([ 2., -2., 9.])
```

Finding a Determinant

The determinant of a square matrix A is often denoted as |A| and is a quantity often used in linear algebra. In SciPy, this is computed using the **det()** function. It takes a matrix as input and returns a scalar value.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
A = np.array([[1,2],[3,4]])

#Passing the values to the det function
x = linalg.det(A)
```

```
#printing the result
print x
```

The above program will generate the following output.

```
-2.0
```

Eigenvalues and Eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. We can find the Eigen values (λ) and the corresponding Eigen vectors (v) of a square matrix (A) by considering the following relation −

$$\mathbf{Av = \lambda v}$$

scipy.linalg.eig computes the eigenvalues from an ordinary or generalized eigenvalue problem. This function returns the Eigen values and the Eigen vectors.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
A = np.array([[1,2],[3,4]])

#Passing the values to the eig function
l, v = linalg.eig(A)

#printing the result for eigen values
print l

#printing the result for eigen vectors
print v
```

The above program will generate the following output.

```
array([-0.37228132+0.j, 5.37228132+0.j]) #--Eigen Values
array([[-0.82456484, -0.41597356], #--Eigen Vectors
       [ 0.56576746, -0.90937671]])
```

SciPy- Ndimage

The SciPy ndimage submodule is dedicated to image processing. Here, ndimage means an n-dimensional image.

Some of the most common tasks in image processing are as follows &miuns;

- Input/Output, displaying images
- Basic manipulations − Cropping, flipping, rotating, etc.
- Image filtering − De-noising, sharpening, etc.
- Image segmentation − Labeling pixels corresponding to different objects
- Classification
- Feature extraction
- Registration

SciPy-Stats

All of the statistics functions are located in the sub-package **scipy.stats** and a fairly complete listing of these functions can be obtained using **info(stats)** function. A list of random variables available can also be obtained from the **docstring** for the stats sub-package. This module contains a large number of probability distributions as well as a growing library of statistical functions.

Normal Continuous Random Variable

A probability distribution in which the random variable X can take any value is continuous random variable. The location (loc) keyword specifies the mean. The scale (scale) keyword specifies the standard deviation.

As an instance of the **rv_continuous** class, **norm** object inherits from it a collection of generic methods and completes them with details specific for this particular distribution.

To compute the CDF at a number of points, we can pass a list or a NumPy array. Let us consider the following example.

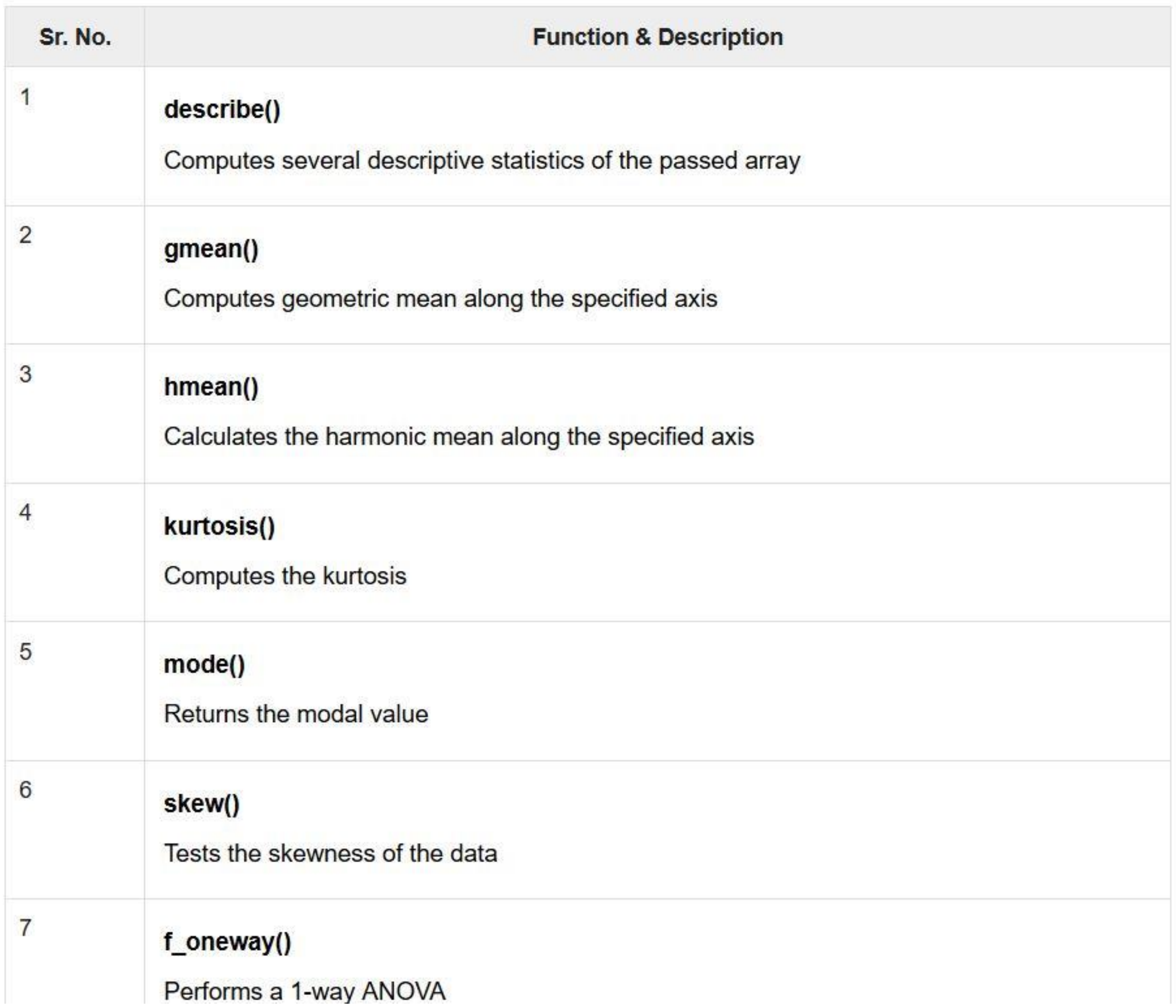
```
from scipy.stats import norm
import numpy as np
print norm.cdf(np.array([1,-1., 0, 1, 3, 4, -2, 6]))
```

The above program will generate the following output.

```
array([ 0.84134475, 0.15865525, 0.5 , 0.84134475, 0.9986501 ,
0.99996833, 0.02275013, 1. ])
```

Descriptive Statistics

The basic stats such as Min, Max, Mean and Variance takes the NumPy array as input and returns the respective results. A few basic statistical functions available in the **scipy.stats package** are described in the following table.



Sr. No.	Function & Description
1	**describe()** Computes several descriptive statistics of the passed array
2	**gmean()** Computes geometric mean along the specified axis
3	**hmean()** Calculates the harmonic mean along the specified axis
4	**kurtosis()** Computes the kurtosis
5	**mode()** Returns the modal value
6	**skew()** Tests the skewness of the data
7	**f_oneway()** Performs a 1-way ANOVA

8	**iqr()** Computes the interquartile range of the data along the specified axis
9	**zscore()** Calculates the z score of each value in the sample, relative to the sample mean and standard deviation
10	**sem()** Calculates the standard error of the mean (or standard error of measurement) of the values in the input array