**MODULE – 2** **Python for Data Science**

**Python for Data Analysis:**

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.
**Numpy** is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python.

**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.

Standard Python distribution doesn't come bundled with NumPy module. A lightweight alternative is to install NumPy using popular Python package installer, **pip**.

**pip install numpy**

NumPy package is imported using the following syntax −

**import numpy as np**

**Arrays in Numpy**

Array in Numpy is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

Number of dimensions of the array is called rank of the array.

A tuple of integers giving the size of the array along each dimension is known as shape of the array.

An array class in Numpy is called as **ndarray**. Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.

**Creating a Numpy Array**

Arrays in Numpy can be created by multiple ways, with various number of Ranks, defining the size of the Array. Arrays can also be created with the use of various data types such as lists, tuples, etc

**Note:** Type of array can be explicitly defined while creating the array.

```
# Python program for  Creation of Arrays
Import numpy as np
 # Creating a rank 1 Array
arr =np.array([1, 2, 3])
print("Array with Rank 1: \n",arr)
```

```
 # Creating a rank 2 Array
arr =np.array([[1, 2, 3],
               [4, 5, 6]])
print("Array with Rank 2: \n", arr)

# Creating an array from tuple
arr =np.array((1, 3, 2))
print("\nArray created using passed tuple:\n", arr)
```

**Output:**

```
Array with Rank 1:
 [1 2 3]
Array with Rank 2:
 [[1 2 3]
 [4 5 6]]
Array created using passed tuple:
 [1 3 2]
```

**Basic Array Operations**

Arrays allow a wide range of operations which can be performed on a particular array or a combination of Arrays. These operations include some basic Mathematical operation as well as Unary and Binary operations.

```
# Python program to demonstrate
# basic operations on single array

Import numpy as np

# Defining Array 1
a =np.array([[1, 2],
             [3, 4]])

# Defining Array 2
b =np.array([[4, 3],
             [2, 1]])

# Adding 1 to every element
print("Adding 1 to every element:", a +1)

# Subtracting 2 from each element
print("\nSubtracting 2 from each element:", b -2)

# sum of array elements
# Performing Unary operations
```

```
print("\nSum of all array "
      "elements: ", a.sum())

# Adding two arrays
# Performing Binary operations
print("\nArray sum:\n", a +b)
```

**Output:**

```
Adding 1 to every element:
 [[2 3]
 [4 5]]
Subtracting 2 from each element:
 [[ 2  1]
 [ 0 -1]]
Sum of all array elements:  10
Array sum:
 [[5 5]
 [5 5]]
```

**Data Types in Numpy**

Every Numpy array is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. Every ndarray has an associated data type (dtype) object. This data type object (dtype) provides information about the layout of the array. The values of an ndarray are stored in a buffer which can be thought of as a contiguous block of memory bytes which can be interpreted by the dtype object.

Numpy provides a large set of numeric datatypes that can be used to construct arrays. At the time of Array creation, Numpy tries to guess a datatype, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

**Constructing a Datatype Object**

Datatypes of Arrays need not to be defined unless a specific datatype is required. Numpy tries to guess the datatype for Arrays which are not predefined in the constructor function.

```
# Python Program to create
# a data type object
Import numpy as np

# Integer datatype
# guessed by Numpy
x =np.array([1, 2])
print("Integer Datatype: ")
```

```
print(x.dtype)

# Float datatype
# guessed by Numpy
x =np.array([1.0, 2.0])
print("\nFloat Datatype: ")
print(x.dtype)

# Forced Datatype
x =np.array([1, 2], dtype =np.int64)
print("\nForcing a Datatype: ")
print(x.dtype)
```

Run on IDE

**Output:**

```
Integer Datatype:

int64

Float Datatype:

float64

Forcing a Datatype:

int64
```

**Math Operations on DataType array**

arrays, basic mathematical operations are performed element-wise on the array. These operations are applied both as operator overloads and as functions. Many useful functions are provided in Numpy for performing computations on Arrays such as **sum**: for addition of Array elements, **T**: for Transpose of elements, etc.

```
# Python Program to create
# a data type object
importnumpy as np

# First Array
arr1 =np.array([[4, 7], [2, 6]],
               dtype =np.float64)

# Second Array
arr2 =np.array([[3, 6], [2, 8]],
               dtype =np.float64)

# Addition of two Arrays
Sum=np.add(arr1, arr2)
print("Addition of Two Arrays: ")
```

```
print(Sum)

# Addition of all Array elements
# using predefined sum method
Sum1 =np.sum(arr1)
print("\nAddition of Array elements: ")
print(Sum1)

# Square root of Array
Sqrt =np.sqrt(arr1)
print("\nSquare root of Array1 elements: ")
print(Sqrt)

# Transpose of Array
# using In-built function 'T'
Trans_arr =arr1.T
print("\nTranspose of Array: ")
print(Trans_arr)
```

**Output:**

```
Addition of Two Arrays:

[[ 7. 13.]

 [ 4. 14.]]

Addition of Array elements:

19.0

Square root of Array1 elements:

[[2.        2.64575131]

 [1.41421356 2.44948974]]

Transpose of Array:

[[4. 2.]

 [7. 6.]]
```

The following are various array attributes.

1. ndim: To get dimension of the array
2. shape: To get shape of the array
3. size: To get size of the array which is nothing but number of elements.
4. dtype: To get data type of array elements
5. itemsize: The size of each array element in bytes

Python program to demonstrate
# basic array characteristics
importnumpy as np

```
# Creating array object
arr =np.array( [[ 1, 2, 3],
                [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)

# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)
```

**Output :**

```
Array is of type:  <class 'numpy.ndarray'>

No. of dimensions:  2

Shape of array:  (2, 3)

Size of array:  6

Array stores elements of type:  int64
```

## Indexing

NumPy or Numeric Python is a package for computation on homogenous n-dimensional arrays. In numpy dimensions are called as axes.

**Types of Indexing**

There are two types of indexing:

**Basic Slicing and indexing :** Consider the syntax x[obj] where x is the array and obj is the index. Slice object is the index in case of basic slicing. Basic slicing occurs when obj is :

- a slice object that is of the form start : stop : step
- an integer or a tuple of slice objects and integers

**Access Array Elements**

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[0])
```

**Access 2-D Arrays**

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the row represents the dimension and the index represents the column.

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].
We can also define the step, like this: [*start*:*end*:*step*].
If we don't pass start its considered 0
If we don't pass end its considered length of array in that dimension
If we don't pass step its considered 1

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

## Types of Indexing

There are two types of indexing:

**Basic Slicing and indexing :** Consider the syntax x[obj] where x is the array and obj is the index. Slice object is the index in case of basic slicing. Basic slicing occurs when obj is :

- a slice object that is of the form start : stop : step
- an integer
- or a tuple of slice objects and integers
-

arrays generated by basic slicing are always view of the original array.

```
# Python program for basic slicing.
import numpy as np

# Arrange elements from 0 to 19
a = np.arange(20)
print("\n Array is:\n ",a)

# a[start:stop:step]
print("\n a[-8:17:1]  = ",a[-8:17:1])

# The : operator means all elements till the end.
print("\n a[10:]  = ",a[10:])
```

**Output :**

```
Array is:

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

a[-8:17:1]  =  [12 13 14 15 16]

a[10:] = [10 11 12 13 14 15 16 17 18 19]
```

## Python Pandas Introduction

Pandas are defined as an open-source library that provides high-performance data manipulation in Python. The name of Pandas is derived from the word **Panel Data**, which means **an Econometrics from Multidimensional data**. It is used for data analysis in Python and developed by **Wes McKinney** in **2008**.

Data analysis requires lots of processing, such as **restructuring, cleaning** or **merging**, etc. There are different tools are available for fast data processing, such as **Numpy, Scipy, Cython**, and **Panda**. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.

Pandas is built on top of the **Numpy** package, means **Numpy** is required for operating the Pandas. Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., **load, manipulate, prepare, model, and analyze**.

**Key Features of Pandas**

- It has a fast and efficient DataFrame object with the default and customized indexing.
- Used for reshaping and pivoting of the data sets.
- Group by data for aggregations and transformations.
- It is used for data alignment and integration of the missing data.
- Provide the functionality of Time Series.
- Process a variety of data sets in different formats like matrix data, tabular heterogeneous, time series.
- Handle multiple operations of the data sets such as subsetting, slicing, filtering, groupBy, re-ordering, and re-shaping.
- It integrates with the other libraries such as SciPy, and scikit-learn.
- Provides fast performance, and If you want to speed it, even more, you can use the **Cython**.

**Benefits of Pandas**

The benefits of pandas over using other language are as follows:

- **Data Representation:** It represents the data in a form that is suited for data analysis through its DataFrame and Series.

- **Clear code:** The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

The Pandas provides two data structures for processing the data,
i.e., **Series** and **DataFrame**,

**1) Series**

It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the **index**. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:

**pandas.Series**

A pandas Series can be created using the following constructor −

pandas.Series( data, index, dtype, copy)

| Sr.No | Parameter & Description |
|---|---|
| 1 | **data**<br>data takes various forms like ndarray, list, constants |
| 2 | **index**<br>Index values must be unique and hashable, same length as data. Default **np.arrange(n)** if no index is passed. |
| 3 | **dtype**<br>dtype is for data type. If None, data type will be inferred |
| 4 | **copy**<br>Copy data. Default False |

**Data:** It can be any list, dictionary, or scalar value.

**Creating Series from Array:**
Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program

```
import pandas as pd
import numpy as np
info = np.array(['P','a','n','d','a','s'])
a = pd.Series(info)
print(a)
```

**Output**

```
0  P
1  a
2  n
3  d
4  a
5  s
dtype: object
```

***Example 2***

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print s
```

Its **output** is as follows −

```
100 a
101 b
102 c
103 d
dtype: object
```

**Create a Series from dict**

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

**Example 1**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data ={'a':0.,'b':1.,'c':2.}
s = pd.Series(data)
print s
```

Its **output** is as follows −

```
a 0.0
b 1.0
c 2.0
dtype: float64
```

**Observe** − Dictionary keys are used to construct index.

**Example 2**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data ={'a':0.,'b':1.,'c':2.}
```

```
s = pd.Series(data,index=['b','c','d','a'])
print s
```

Its **output** is as follows −

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

**Observe** − Index order is persisted and the missing element is filled with NaN (Not a Number).

**Create a Series from Scalar**

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0,1,2,3])
print s
```

Its **output** is as follows −

```
0 5
1 5
2 5
3 5
dtype: int64
```

**Accessing Data from Series with Position**

Data in the series can be accessed similar to that in an **ndarray.**

**Example 1**

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zero$^{th}$ position and so on.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index =['a','b','c','d','e'])

#retrieve the first element
print s[0]
```

Its **output** is as follows −

```
1
```

## DataFrame

It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index. It consists of the following properties:

- The columns can be heterogeneous types like int, bool, and so on.
- It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor −

**pandas.DataFrame( data, index, columns, dtype, copy)**

The parameters of the constructor are as follows −

| Sr.No | Parameter & Description |
|---|---|
| 1 | **data**<br>data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame. |
| 2 | **index**<br>For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed. |
| 3 | **columns**<br>For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed. |
| 4 | **dtype**<br>Data type of each column. |
| 5 | **copy**<br>This command (or whatever it is) is used for copying of data, if the default is False. |

**Create DataFrame**

A pandas DataFrame can be created using various inputs like −

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

**Create a DataFrame using List:**

We can easily create a DataFrame in Pandas using list.

```
import pandas as pd
# a list of strings
x = ['Python', 'Pandas']

# Calling DataFrame constructor on list
df = pd.DataFrame(x)
print(df)
```

**Output**

```
     0
0  Python
1  Pandas
```

**Explanation:** In this code, we have defined a variable named "x" that consist of string values. The DataFrame constructor is being called on a list to print the values.

**Create a DataFrame from Dict of ndarrays / Lists**

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

**Example 1**

```
import pandas as pd
data ={'Name':['Tom','Jack','Steve','Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print df
```

Its **output** is as follows −

```
Age Name
0 28 Tom
```

1 34 Jack
2 29 Steve
3 42 Ricky

**Note** − Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

**Example 2**

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data ={'Name':['Tom','Jack','Steve','Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print df
```

Its **output** is as follows −

```
Age Name
rank1 28 Tom
rank2 34 Jack
rank3 29 Steve
rank4 42 Ricky
```

**Create a DataFrame from List of Dicts**

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

**Example 1**

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd
data =[{'a':1,'b':2},{'a':5,'b':10,'c':20}]
df = pd.DataFrame(data)
print df
```

Its **output** is as follows −

```
   a b c
0 1 2 NaN
1 5 10 20.0
```

**Note** − Observe, NaN (Not a Number) is appended in missing areas.

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd
data =[{'a':1,'b':2},{'a':5,'b':10,'c':20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first','second'], columns=['a','b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first','second'], columns=['a','b1'])
print df1
print df2
```

Its **output** is as follows −

```
#df1 output
a b
first 1 2
second 5 10

#df2 output
a b1
first 1 NaN
second 5 NaN
```

**Note** − Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

**Example**

```
import pandas as pd
d ={'one': pd.Series([1,2,3], index=['a','b','c']), 'two': pd.Series([1,2,3,4], index=['a','b','c','d'])}

df = pd.DataFrame(d)
print df
```

Its **output** is as follows −

```
one two
a 1.0 1
b 2.0 2
c 3.0 3
d NaN 4
```

**Missing Data**

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

```
# import the pandas library
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f','h'],
columns=['one','two','three'])
df = df.reindex(['a','b','c','d','e','f','g','h'])
print (df)
```

**output:**

```
one two three
a 0.077988 0.476149 0.965836
b NaN NaN NaN
c -0.390208 -0.551605 -2.301950
d NaN NaN NaN
e -2.000303 -0.788201 1.510072
f -0.930230 -0.670473 1.146615
g NaN NaN NaN
h 0.085100 0.532791 0.887415
```

Using reindexing, we have created a DataFrame with missing values. In the output, **NaN** means **Not a Number.**

## Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects −

**Example 1**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',
'h'],columns=['one','two','three'])
df = df.reindex(['a','b','c','d','e','f','g','h'])

print (df['one'].isnull())
```

**output** −

```
a False
b True
c False
d True
e False
f False
g True
h False
Name: one, dtype: bool
```

**Example 2**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',
'h'],columns=['one','two','three'])

df = df.reindex(['a','b','c','d','e','f','g','h'])

print (df['one'].notnull())
```

**output** -

```
a True
b False
c True
d False
e True
f True
g False
h True
Name: one, dtype: bool
```

**Calculations with Missing Data**

- When summing data, NA will be treated as Zero
- If the data are all NA, then the result will be NA

**Example 1**

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',
'h'],columns=['one','two','three'])
df = df.reindex(['a','b','c','d','e','f','g','h'])
print (df['one'].sum())
```

**output** –

```
2.02357685917
```

**Example 2**

```
import pandas as pd
```

```
import numpy as np

df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])
print(df)
print (df['one'].sum())
```

**output** −

0

**Cleaning / Filling Missing Data**

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways

**Replace NaN with a Scalar Value**

The following program shows how you can replace "NaN" with "0".

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(3,3), index=['a','c','e'],columns=['one',
'two','three'])

df = df.reindex(['a','b','c'])

print(df)
print("NaN replaced with '0':")
print( df.fillna(0))
```

Its **output** is as follows −

```
one two three
a -0.576991 -0.741695 0.553172
b NaN NaN NaN
c 0.744328 -1.735166 1.749580

NaN replaced with '0':
one two three
a -0.576991 -0.741695 0.553172
b 0.000000 0.000000 0.000000
c 0.744328 -1.735166 1.749580
```

Here, we are filling with value zero; instead we can also fill with any other value.

**Fill NA Forward and Backward**

Using the concepts of filling discussed in the ReIndexing Chapter we will fill the missing values.

| Sr.No | Method & Action |
|---|---|

| | |
|---|---|
| 1 | **pad/fill**<br>Fill methods Forward |
| 2 | **bfill/backfill**<br>Fill methods Backward |

**Example 1**

```
import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',

'h'],columns=['one','two','three'])

df = df.reindex(['a','b','c','d','e','f','g','h'])

print(df)

print("After filling :")

print (df.fillna(method='pad'))
```

Its **output** is as follows −

```
one two three
a 0.077988 0.476149 0.965836
b 0.077988 0.476149 0.965836
c -0.390208 -0.551605 -2.301950
d -0.390208 -0.551605 -2.301950
e -2.000303 -0.788201 1.510072
f -0.930230 -0.670473 1.146615
g -0.930230 -0.670473 1.146615
h 0.085100 0.532791 0.887415
```

**Example 2**

```
import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',

'h'],columns=['one','two','three'])

df = df.reindex(['a','b','c','d','e','f','g','h'])

print(df)

print("After filling :")

print(df.fillna(method='backfill'))
```

Its **output** is as follows −

```
one two three
a 0.077988 0.476149 0.965836
b -0.390208 -0.551605 -2.301950
c -0.390208 -0.551605 -2.301950
d -2.000303 -0.788201 1.510072
e -2.000303 -0.788201 1.510072
f -0.930230 -0.670473 1.146615
g 0.085100 0.532791 0.887415
h 0.085100 0.532791 0.887415
```

**Drop Missing Values**

If you want to simply exclude the missing values, then use the **dropna** function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

**Example 1**

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',
'h'],columns=['one','two','three'])

df = df.reindex(['a','b','c','d','e','f','g','h'])
print df.dropna()
```

Its **output** is as follows –

```
one two three
a 0.077988 0.476149 0.965836
c -0.390208 -0.551605 -2.301950
e -2.000303 -0.788201 1.510072
f -0.930230 -0.670473 1.146615
h 0.085100 0.532791 0.887415
```

**Example 2**

```
import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=['a','c','e','f',

'h'],columns=['one','two','three'])

df = df.reindex(['a','b','c','d','e','f','g','h'])

print (df)

print (df.dropna(axis=1))
```

Its **output** is as follows –

```
  one       two     three
a  1.230344 -1.068563  0.016282
```

```
b       NaN       NaN       NaN
c -0.333450  0.574993  0.653371
d       NaN       NaN       NaN
e -1.502007 -1.395659 -0.998385
f  1.783834 -0.293311 -1.625290
g       NaN       NaN       NaN
h -1.139902  1.137951  1.285422

Empty DataFrame
Columns: [ ]
Index: [a, b, c, d, e, f, g, h]
```

**Replace Missing (or) Generic Values**

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the **fillna()** function.

**Example 1**

```
import pandas as pd

import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000],'two':[1000,0,30,40,50,60]})

print (df.replace({1000:10,2000:60}))
```

Its **output** is as follows −

```
one two
0 10 10
1 20 0
2 30 30
3 40 40
4 50 50
5 60 60
```

# groupby

Any **groupby** operation involves one of the following operations on the original object. They are −

- **Splitting** the Object
- **Applying** a function
- **Combining** the results

In many situations, we split the data into sets and we apply some functionality on each subset. In the apply functionality, we can perform the following operations −

- **Aggregation** − computing a summary statistic
- **Transformation** − perform some group-specific operation
- **Filtration** − discarding the data with some condition

```
#import the pandas library
import pandas as pd

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)

printdf
```

Its **output** is as follows −

```
Points Rank Team Year
0 876 1 Riders 2014
1 789 2 Riders 2015
2 863 2 Devils 2014
3 673 3 Devils 2015
4 741 3 Kings 2014
5 812 4 kings 2015
6 756 1 Kings 2016
7 788 1 Kings 2017
8 694 2 Riders 2016
9 701 4 Royals 2014
10 804 1 Royals 2015
11 690 2 Riders 2017
```

**Split Data into Groups**

Pandas object can be split into any of their objects. There are multiple ways to split an object like −

- obj.groupby('key')
- obj.groupby(['key1','key2'])
- obj.groupby(key,axis=1)

Let us now see how the grouping objects can be applied to the DataFrame object

**Example**

```
# import the pandas library
import pandas as pd

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)

printdf.groupby('Team')
```

Its **output** is as follows −

```
<pandas.core.groupby.DataFrameGroupBy object at 0x7fa46a977e50>
```

**View Groups**

```
# import the pandas library

    import pandas as pd

    ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
    'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
    'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
    'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
    'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
    df=pd.DataFrame(ipl_data)

    print(df.groupby('Team').groups)
```

Its **output** is as follows −

**{'Devils': [2, 3], 'Kings': [4, 5, 6, 7], 'Riders': [0, 1, 8, 11], 'Royals': [9, 10]}**

**Example**

**Group by** with multiple columns −

```
# import the pandas library
import pandas as pd

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)

printdf.groupby(['Team','Year']).groups
```

Its **output** is as follows −

```
{('Devils', 2014): [2],
('Devils', 2015): [3],
 ('Kings', 2014): [4],
('Kings', 2015): [5],
('Kings', 2016): [6],
 ('Kings', 2017): [7],
 ('Riders', 2014): [0],
('Riders', 2015): [1],
 ('Riders', 2016): [8],
('Riders', 2017): [11],
 ('Royals', 2014): [9],
('Royals', 2015): [10]}
```

**Iterating through Groups**

With the **groupby** object in hand, we can iterate through the object similar to itertools.obj.

```
# import the pandas library
import pandas as pd

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)
grouped =df.groupby('Year')

forname,groupin grouped:
print( name)
print (group)
```

Its **output** is as follows −

2014
Points Rank Team Year
0 876 1 Riders 2014
2 863 2 Devils 2014
4 741 3 Kings 2014
9 701 4 Royals 2014

2015
Points Rank Team Year
1 789 2 Riders 2015
3 673 3 Devils 2015
5 812 4 kings 2015
10 804 1 Royals 2015

2016
Points Rank Team Year
6 756 1 Kings 2016
8 694 2 Riders 2016

2017
Points Rank Team Year
7 788 1 Kings 2017
11 690 2 Riders 2017

By default, the **groupby** object has the same label name as the group name.

**Select a Group**

Using the **get_group()** method, we can select a single group.

```
# import the pandas library
        import pandas as pd

        ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
        'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
        'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
        'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
        'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
        df=pd.DataFrame(ipl_data)

        grouped =df.groupby('Year')
        printgrouped.get_group(2014)
```

Its **output** is as follows −

Points Rank Team Year
0 876 1 Riders 2014
2 863 2 Devils 2014
4 741 3 Kings 2014
9 701 4 Royals 2014

**Aggregations**

An aggregated function returns a single aggregated value for each group. Once the **group by** object is created, several aggregation operations can be performed on the grouped data.

An obvious one is aggregation via the aggregate or equivalent **agg** method −

```
# import the pandas library
import pandas as pd
importnumpyas np

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)

grouped =df.groupby('Year')
print grouped['Points'].agg(np.mean)
```

Its **output** is as follows −

```
Year
2014 795.25
2015 769.50
2016 725.00
2017 739.00
Name: Points, dtype: float64
```

Another way to see the size of each group is by applying the size() function −

```
import pandas as pd
importnumpyas np

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)

AttributeAccessinPythonPandas
grouped =df.groupby('Team')
printgrouped.agg(np.size)
```

Its **output** is as follows −

```
Team
Devils    2
```

```
Kings    4
Riders   4
Royals   2
Name: Points, dtype: int64
```

**Applying Multiple Aggregation Functions at Once**

With grouped Series, you can also pass a **list** or **dict of functions** to do aggregation with, and generate DataFrame as output −

```
import pandas as pd
importnumpyas np

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,20
16,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)

grouped =df.groupby('Team')
print grouped['Points'].agg([np.sum,np.mean,np.std])
```

Its **output** is as follows −

```
Team sum mean std
Devils 1536 768.000000 134.350288
Kings 2285 761.666667 24.006943
Riders 3049 762.250000 88.567771
Royals 1505 752.500000 72.831998
kings 812 812.000000 NaN
```

**Transformations**

Transformation on a group or a column returns an object that is indexed the same size of that is being grouped. Thus, the transform should return a result that is the same size as that of a group chunk.

```
# import the pandas library
import pandas as pd
import numpy as np

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)
```

```
print(df)
grouped = df.groupby('Team')
# score =lambda x:(x - x.mean())/ x.std()*10
print(grouped.transform(sum))
```

Its **output** is as follows −

| | Rank | Year | Points |
|---|---|---|---|
| 0 | 7 | 8062 | 3049 |
| 1 | 7 | 8062 | 3049 |
| 2 | 5 | 4029 | 1536 |
| 3 | 5 | 4029 | 1536 |
| 4 | 5 | 6047 | 2285 |
| 5 | 4 | 2015 | 812 |
| 6 | 5 | 6047 | 2285 |
| 7 | 5 | 6047 | 2285 |
| 8 | 7 | 8062 | 3049 |
| 9 | 5 | 4029 | 1505 |
| 10 | 5 | 4029 | 1505 |
| 11 | 7 | 8062 | 3049 |

**Filtration**

Filtration filters the data on a defined criteria and returns the subset of data. The **filter()** function is used to filter the data.

```
import pandas as pd
importnumpyas np

ipl_data={'Team':['Riders','Riders','Devils','Devils','Kings',
'kings','Kings','Kings','Riders','Royals','Royals','Riders'],
'Rank':[1,2,2,3,3,4,1,1,2,4,1,2],
'Year':[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df=pd.DataFrame(ipl_data)
printdf.groupby('Team').filter(lambda x:len(x)>=3)
```

Its **output** is as follows −

| | Points | Rank | Team | Year |
|---|---|---|---|---|
| 0 | 876 | 1 | Riders | 2014 |
| 1 | 789 | 2 | Riders | 2015 |
| 4 | 741 | 3 | Kings | 2014 |
| 6 | 756 | 1 | Kings | 2016 |
| 7 | 788 | 1 | Kings | 2017 |
| 8 | 694 | 2 | Riders | 2016 |
| 11 | 690 | 2 | Riders | 2017 |

In the above filter condition, we are asking to return the teams which have participated three or more times in IPL.

# Merging/Joining

Pandas has full-featured, high performance in-memory join operations idiomatically very similar to relational databases like SQL.

Pandas provides a single function, **merge**, as the entry point for all standard database join operations between DataFrame objects −

## pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=True)

Here, we have used the following parameters −

- **left** − A DataFrame object.
- **right** − Another DataFrame object.
- **on** − Columns (names) to join on. Must be found in both the left and right DataFrame objects.
- **left_on** − Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame.
- **right_on** − Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame.
- **left_index** − If **True,** use the index (row labels) from the left DataFrame as its join key(s). In case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame.
- **right_index** − Same usage as **left_index** for the right DataFrame.
- **how** − One of 'left', 'right', 'outer', 'inner'. Defaults to inner. Each method has been described below.
- **sort** − Sort the result DataFrame by the join keys in lexicographical order. Defaults to True, setting to False will improve the performance substantially in many cases.

Let us now create two different DataFrames and perform the merging operations on it.

```
# import the pandas library
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame(
{'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
print left
print right
```

Its **output** is as follows −

```
Name id subject_id
0 Alex 1 sub1
1 Amy 2 sub2
2 Allen 3 sub4
```

3 Alice 4 sub6
4 Ayoung 5 sub5

Name id subject_id
0 Billy 1 sub2
1 Brian 2 sub4
2 Bran 3 sub3
3 Bryce 4 sub6
4 Betty 5 sub5

**Merge Two DataFrames on a Key**

```
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame({
        'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
Print(pd.merge(left,right,on='id'))
```

Its **output** is as follows −

Name_x id subject_id_xName_ysubject_id_y
0 Alex 1 sub1 Billy sub2
1 Amy 2 sub2 Brian sub4
2 Allen 3 sub4 Bran sub3
3 Alice 4 sub6 Bryce sub6
4 Ayoung 5 sub5 Betty sub5

**Merge Two DataFrames on Multiple Keys**

```
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame({
        'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
printpd.merge(left,right,on=['id','subject_id'])
```

Its **output** is as follows −

Name_x id subject_idName_y
0 Alice 4 sub6 Bryce
1 Ayoung 5 sub5 Betty

**Merge Using 'how' Argument**

The **how** argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or the right tables, the values in the joined table will be NA.

Here is a summary of the **how** options and their SQL equivalent names −

| Merge Method | SQL Equivalent | Description |
|---|---|---|
| left | LEFT OUTER JOIN | Use keys from left object |
| right | RIGHT OUTER JOIN | Use keys from right object |
| outer | FULL OUTER JOIN | Use union of keys |
| inner | INNER JOIN | Use intersection of keys |

**Left Join**

```
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
printpd.merge(left, right, on='subject_id', how='left')
```

Its **output** is as follows −

```
Name_x  id_x  subject_id  Name_y  id_y
0 Alex 1 sub1 NaNNaN
1 Amy 2 sub2 Billy 1.0
2 Allen 3 sub4 Brian 2.0
3 Alice 4 sub6 Bryce 4.0
4 Ayoung 5 sub5 Betty 5.0
```

**Right Join**

```
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
```

```
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
printpd.merge(left, right, on='subject_id', how='right')
```

Its **output** is as follows −

```
Name_xid_xsubject_idName_yid_y
0 Amy 2.0 sub2 Billy 1
1 Allen 3.0 sub4 Brian 2
2 Alice 4.0 sub6 Bryce 4
3 Ayoung 5.0 sub5 Betty 5
4 NaNNaN sub3 Bran 3
```

**Outer Join**

```
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
printpd.merge(left, right, how='outer', on='subject_id')
```

Its **output** is as follows −

```
Name_xid_xsubject_idName_yid_y
0 Alex 1.0 sub1 NaNNaN
1 Amy 2.0 sub2 Billy 1.0
2 Allen 3.0 sub4 Brian 2.0
3 Alice 4.0 sub6 Bryce 4.0
4 Ayoung 5.0 sub5 Betty 5.0
5 NaNNaN sub3 Bran 3.0
```

**Inner Join**

Joining will be performed on index. Join operation honors the object on which it is called. So, **a.join(b)** is not equal to **b.join(a)**.

```
import pandas aspd
left =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5']})
right =pd.DataFrame({
'id':[1,2,3,4,5],
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5']})
printpd.merge(left, right, on='subject_id', how='inner')
```

Its **output** is as follows −

Name_xid_xsubject_idName_yid_y
0 Amy 2 sub2 Billy 1
1 Allen 3 sub4 Brian 2
2 Alice 4 sub6 Bryce 4
3 Ayoung 5 sub5 Betty 5

## Concatenation

Pandas provides various facilities for easily combining together **Series, DataFrame**, and **Panel** objects.

### pd.concat(objs,axis=0,join='outer',join_axes=None, ignore_index=False)

- **objs** − This is a sequence or mapping of Series, DataFrame, or Panel objects.
- **axis** − {0, 1, ...}, default 0. This is the axis to concatenate along.
- **join** − {‘inner’, ‘outer’}, default ‘outer’. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- **ignore_index** − boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1.
- **join_axes** − This is the list of Index objects. Specific indexes to use for the other (n-1) axes instead of performing inner/outer set logic.

**Concatenating Objects**

The **concat** function does all of the heavy lifting of performing concatenation operations along an axis. Let us create different objects and do concatenation.

```
import pandas aspd
one =pd.DataFrame({
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5'],
'Marks_scored':[98,90,87,69,78]},
  index=[1,2,3,4,5])

two =pd.DataFrame({
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5'],
'Marks_scored':[89,80,79,97,88]},
  index=[1,2,3,4,5])
printpd.concat([one,two])
```

Its **output** is as follows −

Marks_scored Name subject_id
1 98 Alex sub1
2 90 Amy sub2
3 87 Allen sub4
4 69 Alice sub6

```
5 78 Ayoung sub5
1 89 Billy sub2
2 80 Brian sub4
3 79 Bran sub3
4 97 Bryce sub6
5 88 Betty sub5
```

Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this by using the **keys** argument −

```
import pandas aspd

one =pd.DataFrame({
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5'],
'Marks_scored':[98,90,87,69,78]},
  index=[1,2,3,4,5])

two =pd.DataFrame({
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5'],
'Marks_scored':[89,80,79,97,88]},
  index=[1,2,3,4,5])
printpd.concat([one,two],keys=['x','y'])
```

Its **output** is as follows −

```
x 1 98 Alex sub1
2 90 Amy sub2
3 87 Allen sub4
4 69 Alice sub6
5 78 Ayoung sub5
y 1 89 Billy sub2
2 80 Brian sub4
3 79 Bran sub3
4 97 Bryce sub6
5 88 Betty sub5
```

The index of the resultant is duplicated; each index is repeated.

If the resultant object has to follow its own indexing, set **ignore_index** to **True**.

```
import pandas aspd

one =pd.DataFrame({
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5'],
'Marks_scored':[98,90,87,69,78]},
  index=[1,2,3,4,5])

two =pd.DataFrame({
```

```
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5'],
'Marks_scored':[89,80,79,97,88]},
  index=[1,2,3,4,5])
printpd.concat([one,two],keys=['x','y'],ignore_index=True)
```

Its **output** is as follows −

```
Marks_scored Name subject_id
0 98 Alex sub1
1 90 Amy sub2
2 87 Allen sub4
3 69 Alice sub6
4 78 Ayoung sub5
5 89 Billy sub2
6 80 Brian sub4
7 79 Bran sub3
8 97 Bryce sub6
9 88 Betty sub5
```

Observe, the index changes completely and the Keys are also overridden.

If two objects need to be added along **axis=1**, then the new columns will be appended.

```
import pandas aspd

one =pd.DataFrame({
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5'],
'Marks_scored':[98,90,87,69,78]},
  index=[1,2,3,4,5])

two =pd.DataFrame({
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5'],
'Marks_scored':[89,80,79,97,88]},
  index=[1,2,3,4,5])
printpd.concat([one,two],axis=1)
```

Its **output** is as follows −

```
Marks_scored Name subject_idMarks_scored Name subject_id
1 98 Alex sub1 89 Billy sub2
2 90 Amy sub2 80 Brian sub4
3 87 Allen sub4 79 Bran sub3
4 69 Alice sub6 97 Bryce sub6
5 78 Ayoung sub5 88 Betty sub5
```

**Concatenating Using append**

A useful shortcut to concat are the append instance methods on Series and DataFrame. These methods actually predated concat. They concatenate along **axis=0**, namely the index –

```
import pandas aspd

one =pd.DataFrame({
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5'],
'Marks_scored':[98,90,87,69,78]},
  index=[1,2,3,4,5])

two =pd.DataFrame({
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5'],
'Marks_scored':[89,80,79,97,88]},
  index=[1,2,3,4,5])
printone.append(two)
```

Its **output** is as follows –

```
Marks_scored Name subject_id
1 98 Alex sub1
2 90 Amy sub2
3 87 Allen sub4
4 69 Alice sub6
5 78 Ayoung sub5
1 89 Billy sub2
2 80 Brian sub4
3 79 Bran sub3
4 97 Bryce sub6
5 88 Betty sub5
```

The **append** function can take multiple objects as well –

```
import pandas aspd

one =pd.DataFrame({
'Name':['Alex','Amy','Allen','Alice','Ayoung'],
'subject_id':['sub1','sub2','sub4','sub6','sub5'],
'Marks_scored':[98,90,87,69,78]},
  index=[1,2,3,4,5])

two =pd.DataFrame({
'Name':['Billy','Brian','Bran','Bryce','Betty'],
'subject_id':['sub2','sub4','sub3','sub6','sub5'],
```

```
'Marks_scored':[89,80,79,97,88]},
  index=[1,2,3,4,5])
printone.append([two,one,two])
```

Its **output** is as follows −

```
Marks_scored Name subject_id
1 98 Alex sub1
2 90 Amy sub2
3 87 Allen sub4
4 69 Alice sub6
5 78 Ayoung sub5
1 89 Billy sub2
2 80 Brian sub4
3 79 Bran sub3
4 97 Bryce sub6
5 88 Betty sub5
1 98 Alex sub1
2 90 Amy sub2
3 87 Allen sub4
4 69 Alice sub6
5 78 Ayoung sub5
1 89 Billy sub2
2 80 Brian sub4
3 79 Bran sub3
4 97 Bryce sub6
5 88 Betty sub5
```

**Pandas read_csv()**

Most of the data is available in a tabular format of CSV files. It is very popular. You can convert them to a pandas DataFrame using the read_csv function.
The pandas.read_csv is used to load a CSV file as a pandas dataframe.

- **Syntax:** pandas.read_csv( filepath_or_buffer, sep, header, index_col, usecols, prefix, dtype, converters, skiprows, skiprows, nrows, na_values, parse_dates)Purpose: Read a comma-separated values (csv) file into DataFrame. Also supports optionally iterating or breaking the file into chunks.
- **Parameters:**
    - **filepath_or_buffer** *:str, path object or file-like object* Any valid string path is acceptable. The string could be a URL too. Path object refers to os.PathLike. File-like objects with a read() method, such as a filehandle (e.g. via built-in open function) or StringIO.
    - **sep** *:str, (Default ‘,’)* Separating boundary which distinguishes between any two subsequent data items.
    - **header** *:int, list of int, (Default ‘infer’)* Row number(s) to use as the column names, and the start of the data. The default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file.
- **names***: array-like* List of column names to use. If the file contains a header row, then you should explicitly pass header=0 to override the column names. Duplicates in this list are not allowed.
- **index_col***: int, str, sequence of int/str, or False, (Default None)* Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int/str is given, a MultiIndex is used.
- **usecols***: list-like or callable* Return a subset of the columns. If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True.
- **prefix***: str* Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1

- **dtype***: Type name or dict of column -> type* Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use str or object together with suitable na_values settings to preserve and not interpret dtype.
- **converters***: dict* Dict of functions for converting values in certain columns. Keys can either be integers or column labels.
- **skiprows***: list-like, int or callable* Line numbers to skip (0-indexed) or the number of lines to skip (int) at the start of the file. If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise.
- **skipfooter***: int* Number of lines at bottom of the file to skip
- **nrows***: int* Number of rows of file to read. Useful for reading pieces of large files.
- **na_values***: scalar, str, list-like, or dict* Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.
- **parse_dates***: bool or list of int or names or list of lists or dict, (default False)* If set to True, will try to parse the index, else parse the columns passed
- **Returns**: DataFrame or TextParser, A comma-separated values (CSV) file is returned as a two-dimensional data structure with labeled axes.

**Reading CSV file**

The pandas read_csv function can be used in different ways as per necessity like using custom separators, reading only selective columns/rows and so on. All cases are covered below one after another.

**Default Separator**

To read a CSV file, call the pandas function read_csv() and pass the file path as input.

**Step 1: Import Pandas**

```
import pandas aspd
```

**Step 2: Read the CSV**

```
# Read the csv file

df=pd.read_csv("data1.csv")

# First 5 rows

df.head()
```

**Different, Custom Separators**

By default, a CSV is seperated by comma. But you can use other seperators as well. The pandas.read_csvfunction is not limited to reading the CSV file with default separator (i.e. comma). It can be used for other separators such as ;, | or :. To load CSV files with such separators, the sep parameter is used to pass the separator used in the CSV file.

```
# Read the csv file sep='|'
df=pd.read_csv("data2.csv",sep='|')
print(df)
```

**Set any row as column header**

```
# Read the csv file
df=pd.read_csv("data1.csv")
df.head()
```

| | Ranking | Name | Count 1 | Count 2 | Dates |
|---|---|---|---|---|---|
| 0 | Rank | State | Population | National Share (%) | Date |
| 1 | 1 | Uttar Pradesh | 19,98,12,341 | 16.51% | 25-02-2021 |
| 2 | 2 | Maharashtra | 11,23,74,333 | 9.28% | 14-04-2021 |
| 3 | 3 | Bihar | 10,40,99,452 | missing | 19-02-2021 |
| 4 | 4 | West Bengal | 9,12,76,115 | 7.54% | 24-02-2021 |

The row 0 seems to be a better fit for the header. It can explain better about the figures in the table. You can make this 0 row as a header while reading the CSV by using the header parameter. Header parameter takes the value as a row number.

**Note: Row numbering starts from 0 including column header**

```
# Read the csv file with header parameter

df=pd.read_csv("data1.csv", header=1)
```

```
df.head()
```

| | Rank | State | Population | National Share (%) | Date |
|---|---|---|---|---|---|
| 0 | 1 | Uttar Pradesh | 19,98,12,341 | 16.51% | 25-02-2021 |
| 1 | 2 | Maharashtra | 11,23,74,333 | 9.28% | 14-04-2021 |
| 2 | 3 | Bihar | 10,40,99,452 | missing | 19-02-2021 |
| 3 | 4 | West Bengal | 9,12,76,115 | 7.54% | 24-02-2021 |
| 4 | 5 | Madhya Pradesh | 7,26,26,809 | 6% | 13-02-2021 |

**Renaming column headers**

While reading the CSV file, you can rename the column headers by using the names parameter. The names parameter takes the list of names of the column header.

```
# Read the csv file with names parameter
df=pd.read_csv("data.csv", names=['Ranking','ST Name','Pop','NS','D'])
df.head()
```

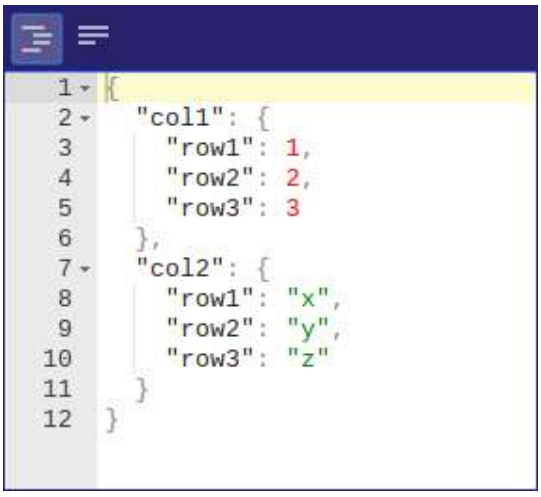
| | Ranking | ST Name | Pop | NS | D |
|---|---|---|---|---|---|
| 0 | Rank | State | Population | National Share (%) | Date |
| 1 | 1 | Uttar Pradesh | 19,98,12,341 | 16.51% | 25-02-2021 |
| 2 | 2 | Maharashtra | 11,23,74,333 | 9.28% | 14-04-2021 |
| 3 | 3 | Bihar | 10,40,99,452 | missing | 19-02-2021 |
| 4 | 4 | West Bengal | 9,12,76,115 | 7.54% | 24-02-2021 |

## Read JSON

**JSON:**

JSON is shorthand for *JavaScript Object Notation*. This is a text format that is often used to exchange data on the web.

The format looks like this:



```
{
  "col1": {
    "row1": 1,
    "row2": 2,
    "row3": 3
  },
  "col2": {
    "row1": "x",
    "row2": "y",
    "row3": "z"
  }
}
```

In practice, this data is often on one line, like so:

{“col1”:{“row1”:1,”row2”:2,”row3”:3},”col2”:{“row1”:”x”,”row2”:”y”,”row3”:”z”}}

Any type of data can be stored in this format (string, integer, float etc).

It’s common for a web server to return and accept json format. This is often how the frontend communicates with the backend.

**pandas.read_json**

The example below parses a JSON string and converts it to a Pandas DataFrame.

```
# load pandas and json modules
import pandas aspd
importjson

# json string
s = '{"col1":{"row1":1,"row2":2,"row3":3},"col2":{"row1":"x","row2":"y","row3":"z"}}'

# read json to data frame
df = pd.read_json(s)
print(df)
```

You can run it to see the output:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | | col1 | col2 | |
| 2 | row1 | 1 | x | |
| 3 | row2 | 2 | y | |
| 4 | row3 | 3 | z | |
| 5 | | | | |
| 6 | | | | |

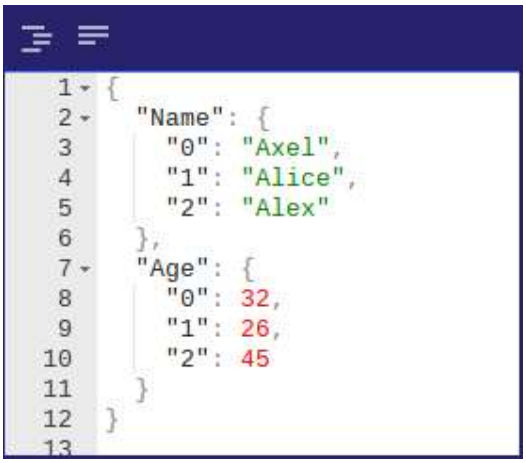
**Save to JSON file**

A *DataFrame* can be saved as a *json file*. To do so, use the method to_json(filename).
If you want to save to a json file, you can do the following:

```
import pandas aspd
importjson
df = pd.DataFrame([1,2,3])
df.to_json('example.json')
```

For a dataframe with several columns:

```
import pandas aspd
importjson
data = [['Axel',32], ['Alice', 26], ['Alex', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
df.to_json('example.json')
```




**Load JSON from File**

If the *json data* is stored in a file, you can load it into a DataFrame.

You can use the example above to create a json file, then use this example to load it into a dataframe.

```
df_f = pd.read_json('files/sample_file.json')
```

**Pandas to JSON example**

In the next example, you load data from a *csv file* into a *dataframe*, that you can then save as *json file*.

You can load a csv file as a pandas dataframe:

```
df = pd.read_csv("data.csv")
```

Then save the **DataFrame to JSON** format:

```
# save a dataframe to json format:
df.to_json("data.json")
```

**Data cleaning:**

When working with multiple data sources, there are many chances for data to be incorrect, duplicated, or mislabeled. If data is wrong, outcomes and algorithms are unreliable, even though they may look correct. *Data cleaning* is the process of changing or eliminating garbage, incorrect, duplicate, corrupted, or incomplete data in a dataset. There's no such absolute way to describe the precise steps in the data cleaning process because the processes may vary from dataset to dataset. Data cleansing, data cleansing, or data scrub is that the initiative among the general data preparation process. Data cleaning plays an important part in developing reliable answers and within the analytical process and is observed to be a basic feature of the info science basics. The motive of data cleaning services is to construct uniform and standardized data sets that enable data analytical tools and business intelligence easy access and perceive accurate data for each problem.

**Why data cleaning is essential?**

Data cleaning is the most important task that should be done as a data science professional. Having wrong or bad quality data can be detrimental to processes and analysis. Having clean data will ultimately increase overall productivity and permit the very best quality information in your decision-making. Following are some reasons why data cleaning is essential:

**Data Cleaning Cycle**

It is the method of analyzing, distinguishing, and correcting untidy, raw data. Data cleaning involves filling in missing values, distinguish and fix errors present in the dataset. Whereas the techniques used for data cleaning might vary in step with different types of datasets, the following are standard steps to map out data cleaning:

# Matplotlib

**matplotlib.pyplot** is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.
Matploitlib is a Python Library used for plotting, this python library provides and objected-oriented APIs for integrating plots into applications.

### Installation of Matplotlib

If you have Python and PIP already installed on a system, then installation of Matplotlib is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install matplotlib
```

### Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding
the import *module* statement:

```
import matplotlib
```

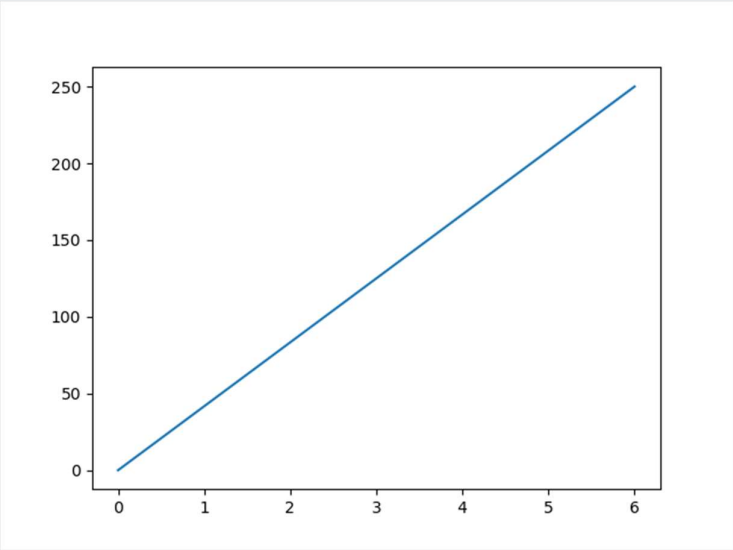
### Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

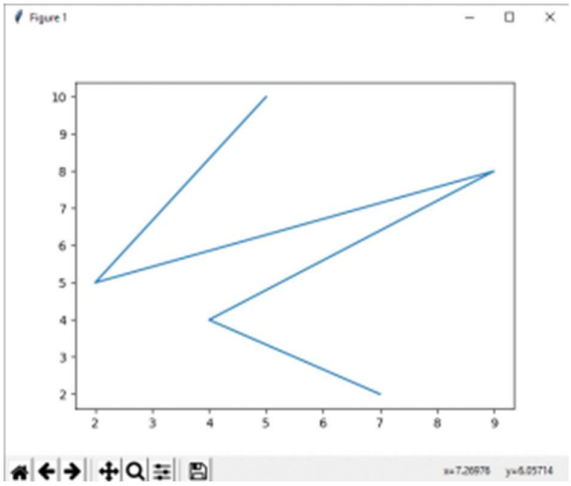
Example

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

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```

**Result:**




```
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 9, 4, 7]

# Y-axis values
y = [10, 5, 8, 4, 2]

# Function to plot
plt.plot(x,y)

# function to show the plot
plt.show()
```




### Plotting x and y points

The plot() function is used to draw points (markers) in a diagram.

By default, the plot() function draws a line from point to point.

The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the **x-axis**.

Parameter 2 is an array containing the points on the **y-axis**

The **x-axis** is the horizontal axis.

The **y-axis** is the vertical axis.

## Plotting Without Line

To plot only the markers, you can use *shortcut string notation* parameter 'o', which means 'rings'.

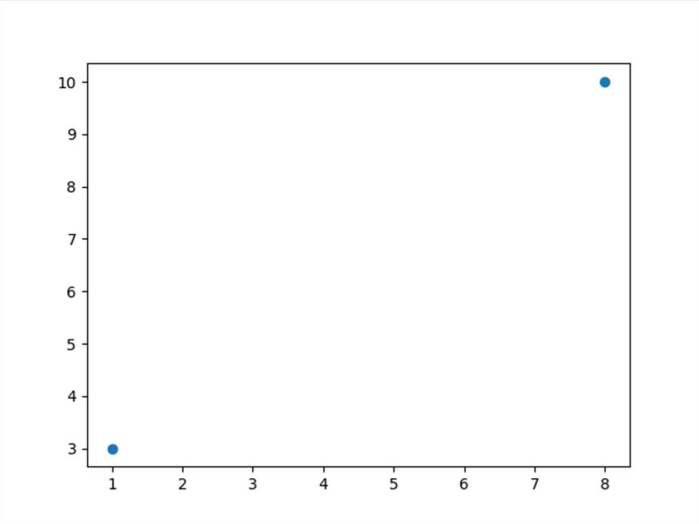
**Example**

Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

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

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])
```

```
plt.plot(xpoints, ypoints, 'o')
plt.show()
```

**Result:**




## **Matplotlib** Markers

Markers

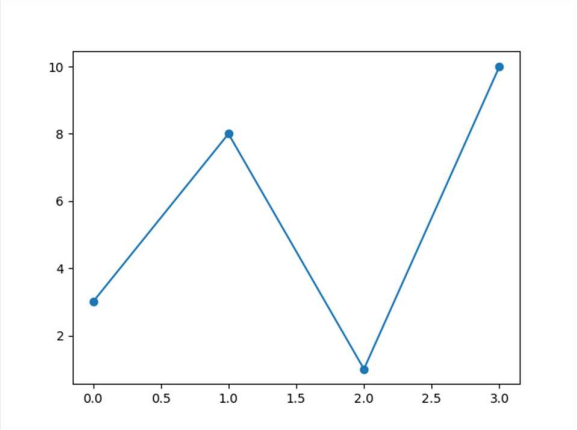
You can use the keyword argument marker to emphasize each point with a specified marker:

**Example**

Mark each point with a circle:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()
```

**Result:**




| | |
|---|---|
| 'o' | Circle |
| '*' | Star |
| '.' | Point |
| ',' | Pixel |
| 'x' | X |

## **Matplotlib** Line

### Linestyle

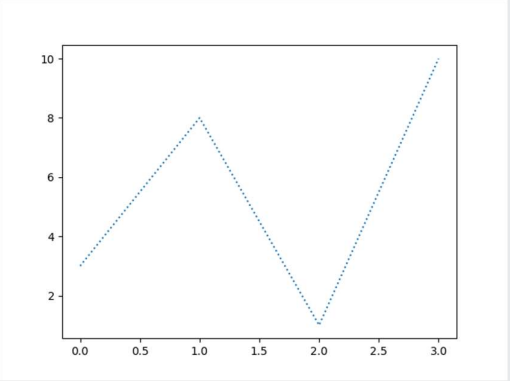
You can use the keyword argument linestyle, or shorter ls, to change the style of the plotted line:

**Example**

Use a dotted line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')
plt.show()
```

**Result:**




```
plt.plot(ypoints, linestyle = 'dashed')
```

## Line Color

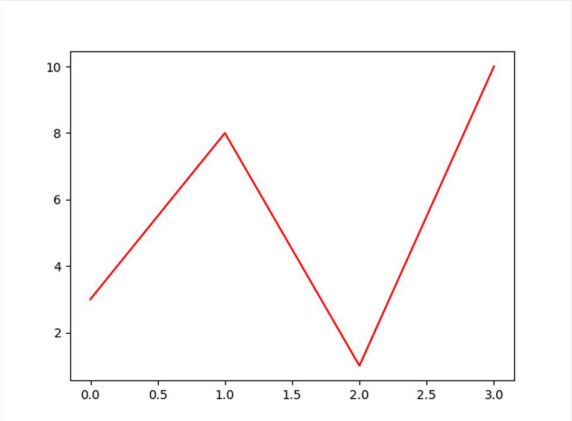
You can use the keyword argument color or the shorter c to set the color of the line:

**Example**

Set the line color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color = 'r')
plt.show()
```

**Result:**



## Line Width

You can use the keyword argument linewidth or the shorter lw to change the width of the line.

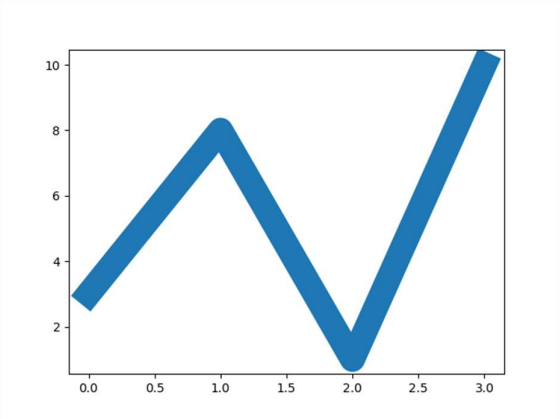
The value is a floating number, in points:

**Example**

Plot with a 20.5pt wide line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```

**Result:**




## Multiple Lines

You can plot as many lines as you like by simply adding more plt.plot() functions:

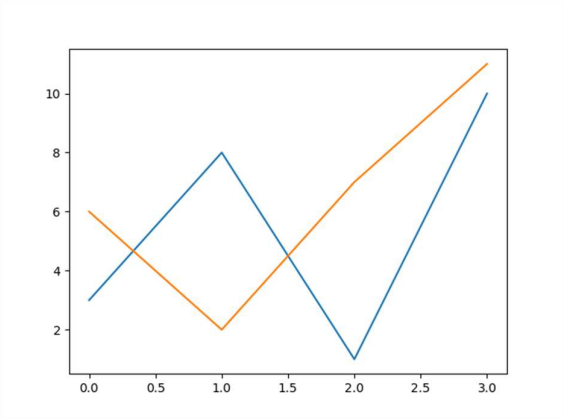
**Example**

Draw two lines by specifying a plt.plot() function for each line:

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

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)
```

```
plt.show()
```

**Result:**




## Create Labels for a Plot

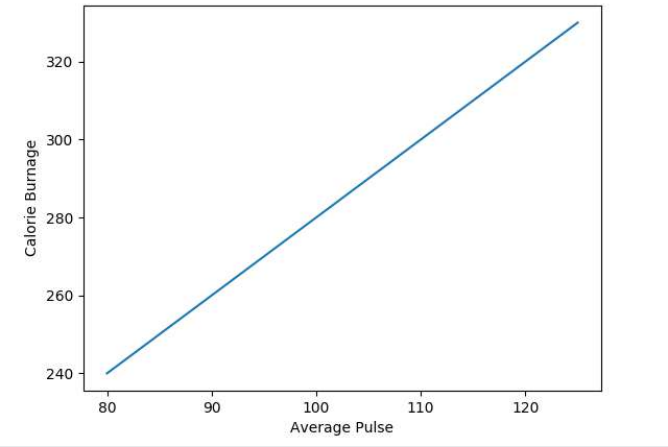
With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

**Example**

Add labels to the x- and y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

**Result:**




## Create a Title for a Plot

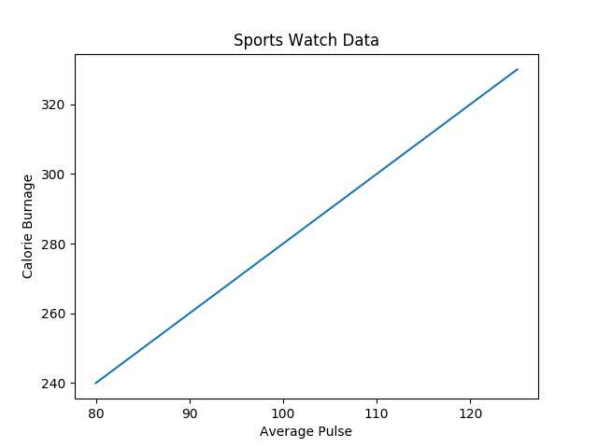
With Pyplot, you can use the title() function to set a title for the plot.

**Example**

Add a plot title and labels for the x- and y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

**Result:**




## Add Grid Lines to a Plot

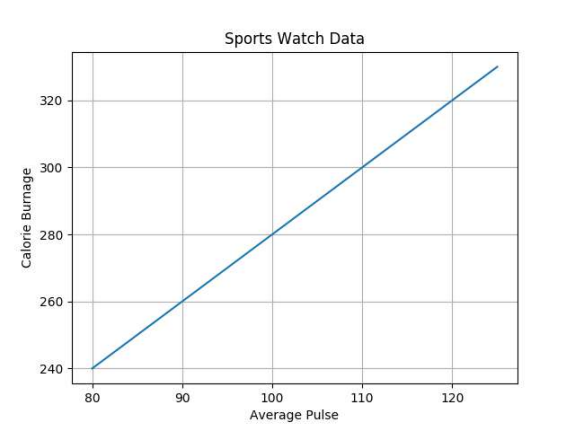
With Pyplot, you can use the grid() function to add grid lines to the plot.

**Example**

Add grid lines to the plot:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.grid()
plt.show()
```

**Result:**




## The subplot() Function

The subplot() function takes three arguments that describes the layout of the figure.

The layout is organized in rows and columns, which are represented by the *first* and *second* argument.

The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.
```

```
plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

So, if we want a figure with 2 rows an 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this:

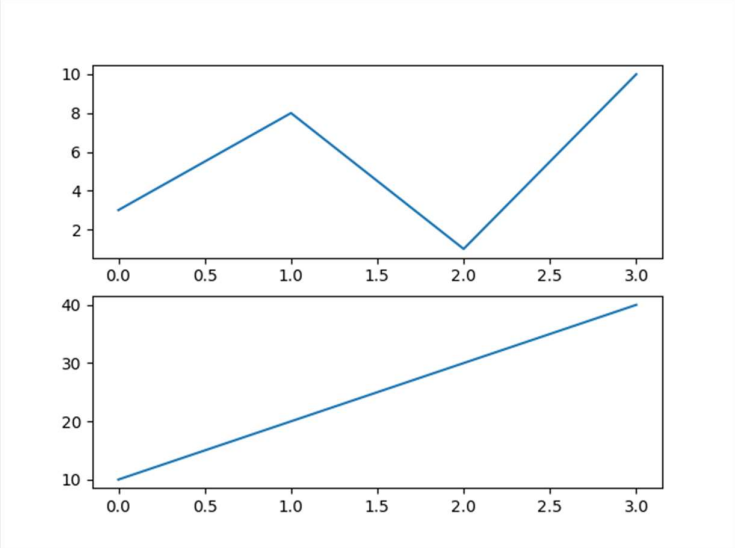
**Example**

Draw 2 plots on top of each other:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 1, 1)
plt.plot(x,y)
```

```
#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 1, 2)
plt.plot(x,y)

plt.show()
```

**Result:**




## Python Matplotlib : Types of Plots

## Creating Scatter Plots

With Pyplot, you can use the scatter() function to draw a scatter plot.

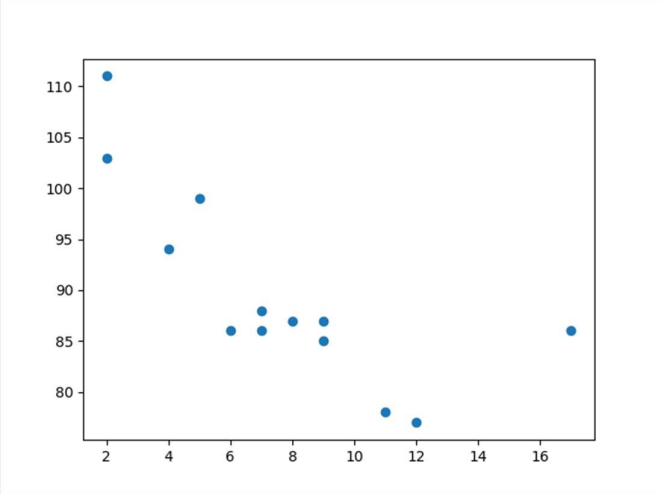
The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

**Example**

A simple scatter plot:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])

plt.scatter(x, y)
plt.show()
```

**Result:**




## Colors

You can set your own color for each scatter plot with the color or the c argument:

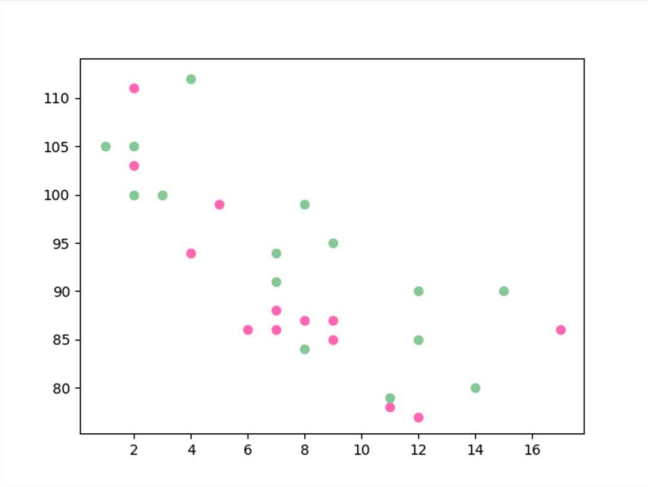
**Example**

Set your own color of the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
```

```
plt.scatter(x, y, color = 'hotpink')

x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```

**Result:**




## Creating Bars

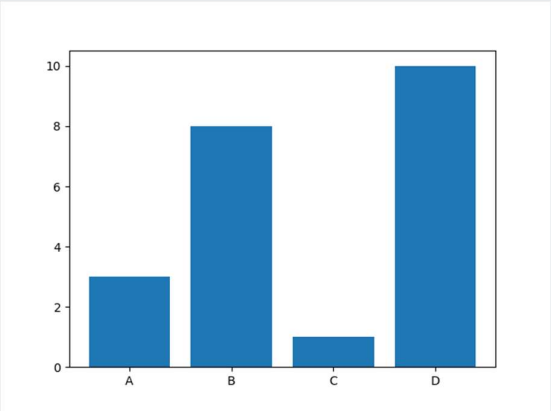
With Pyplot, you can use the bar() function to draw bar graphs:

**Example**

Draw 4 bars:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```

**Result:**




## Horizontal Bars

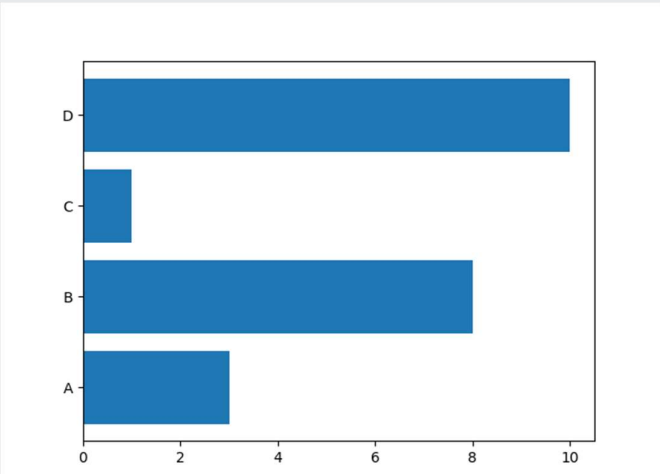
If you want the bars to be displayed horizontally instead of vertically, use the barh() function:

**Example**

Draw 4 horizontal bars:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
plt.barh(x, y)
plt.show()
```

**Result:**



## Bar Color

The bar() and barh() takes the keyword argument color to set the color of the bars:

**Example**

Draw 4 red bars:

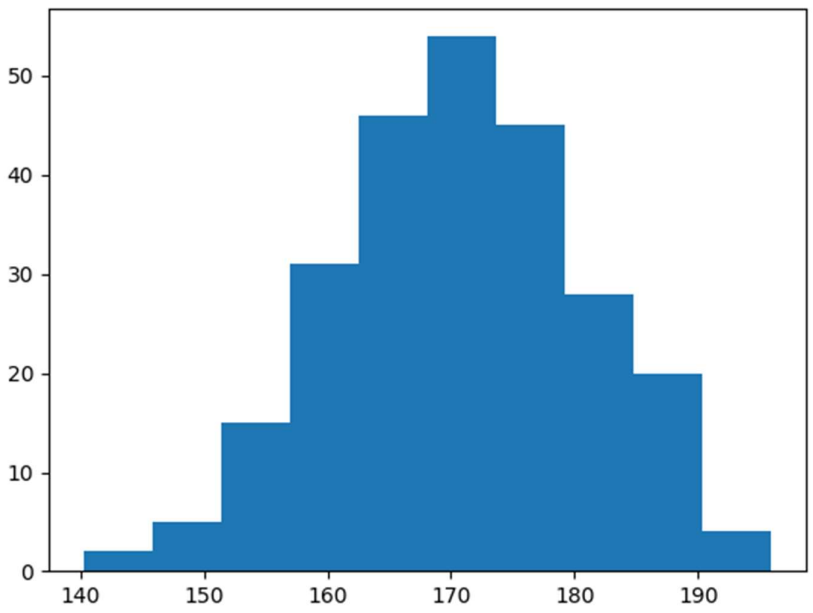
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = "red")
plt.show()
```

# Histogram

A histogram is a graph showing *frequency* distributions.

It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:




You can read from the histogram that there are approximately:

2 people from 140 to 145cm
5 people from 145 to 150cm
15 people from 151 to 156cm
31 people from 157 to 162cm
46 people from 163 to 168cm
53 people from 168 to 173cm
45 people from 173 to 178cm
28 people from 179 to 184cm
21 people from 185 to 190cm
4 people from 190 to 195cm

## Create Histogram

In Matplotlib, we use the hist() function to create histograms.

The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

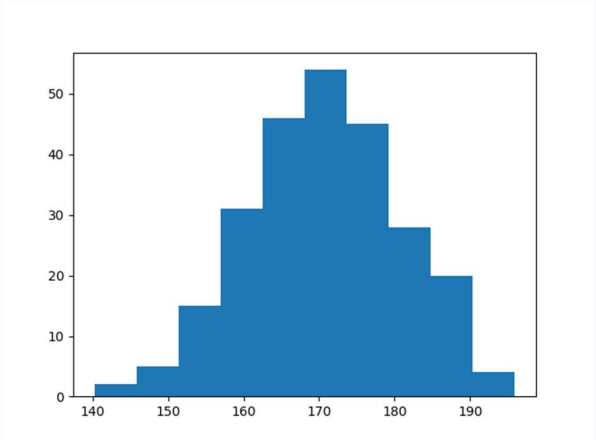
The hist() function will read the array and produce a histogram:

**Example**

A simple histogram:

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

x = np.random.normal(170, 10, 250)

plt.hist(x)
plt.show()
```

**Result:**



## Creating Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

**Example**

A simple pie chart:

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

y = np.array([35, 25, 25, 15])

plt.pie(y)
plt.show()
```

**Result:**

Labels

Add labels to the pie chart with the label parameter.

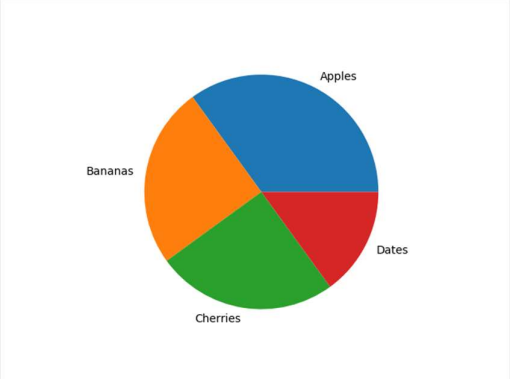
The label parameter must be an array with one label for each wedge:

**Example**

A simple pie chart:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.show()
```

**Result:**




**Colors**

You can set the color of each wedge with the colors parameter.

The colors parameter, if specified, must be an array with one value for each wedge:

**Example**

Specify a new color for each wedge:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
mycolors = ["black", "hotpink", "b", "#4CAF50"]

plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```

**Legend**

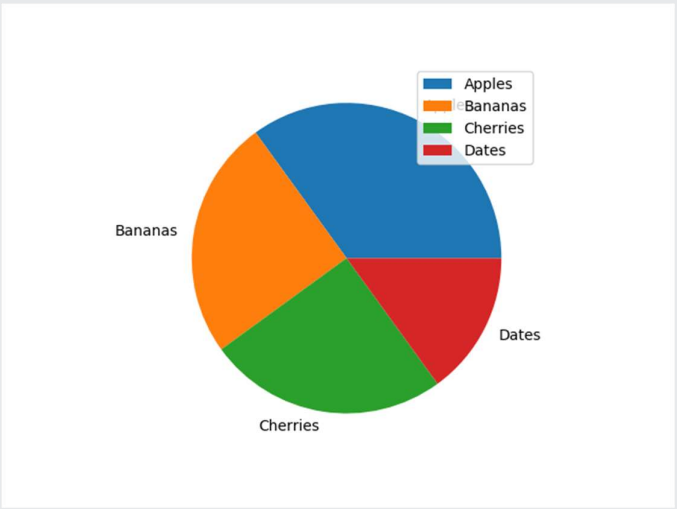
To add a list of explanation for each wedge, use the legend() function:

**Example**

Add a legend:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend()
plt.show()
```

**Result:**



## Seaborn

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

Types of distribution plots namely:

1. joinplot
2. distplot
3. pairplot
4. rugplot

**Displot**

It is used basically for univariant set of observations and visualizes it through a histogram i.e. only one observation and hence we choose one particular column of the dataset.

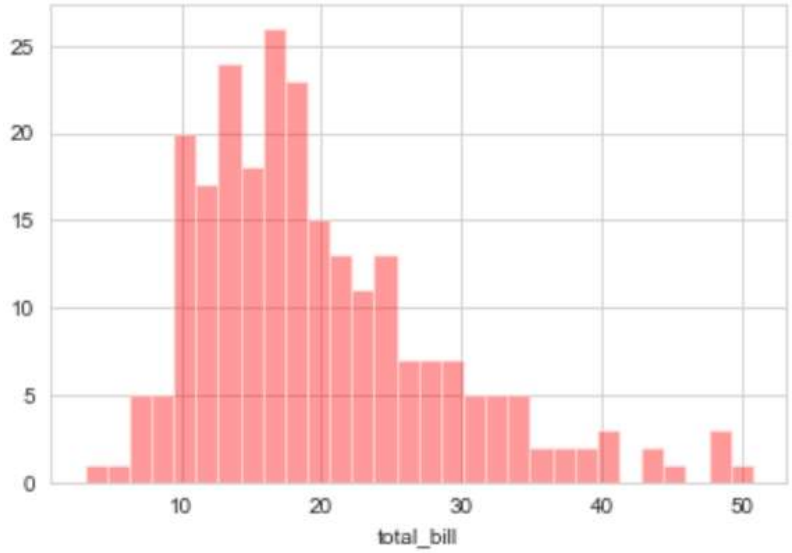
**Syntax:**

distplot(a[, bins, hist, kde, rug, fit, ...])

**Example:**

```
# set the background style of the plot

sns.set_style('whitegrid')

  sns.distplot(df['total_bill'], kde = False, color ='red', bins = 30)
```

**Output:**




**Explanation:**

- KDE stands for **Kernel Density Estimation** and that is another kind of the plot in seaborn.
- bins is used to set the number of bins you want in your plot and it actually depends on your dataset.
- color is used to specify the color of the plot

**Joinplot**

It is used to draw a plot of two variables with bivariate and univariate graphs. It basically combines two different plots.

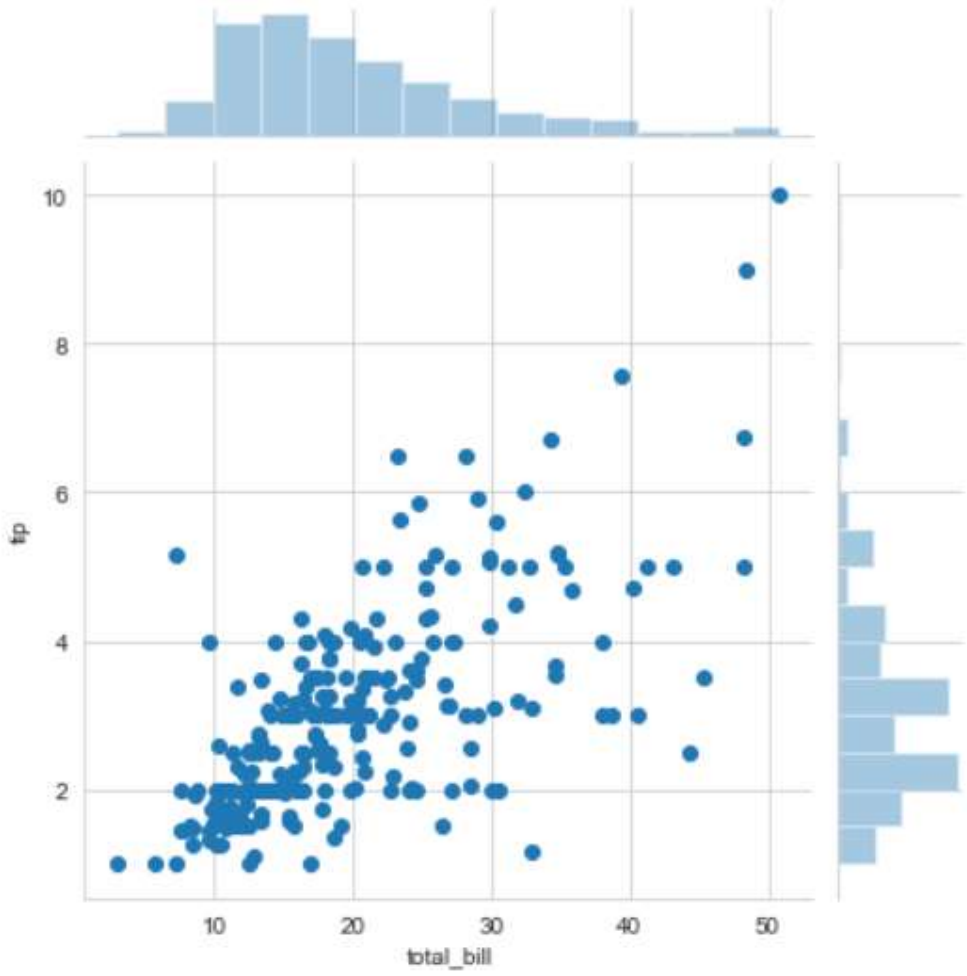
**Syntax:**

jointplot(x, y[, data, kind, stat_func, ...])

**Example:**

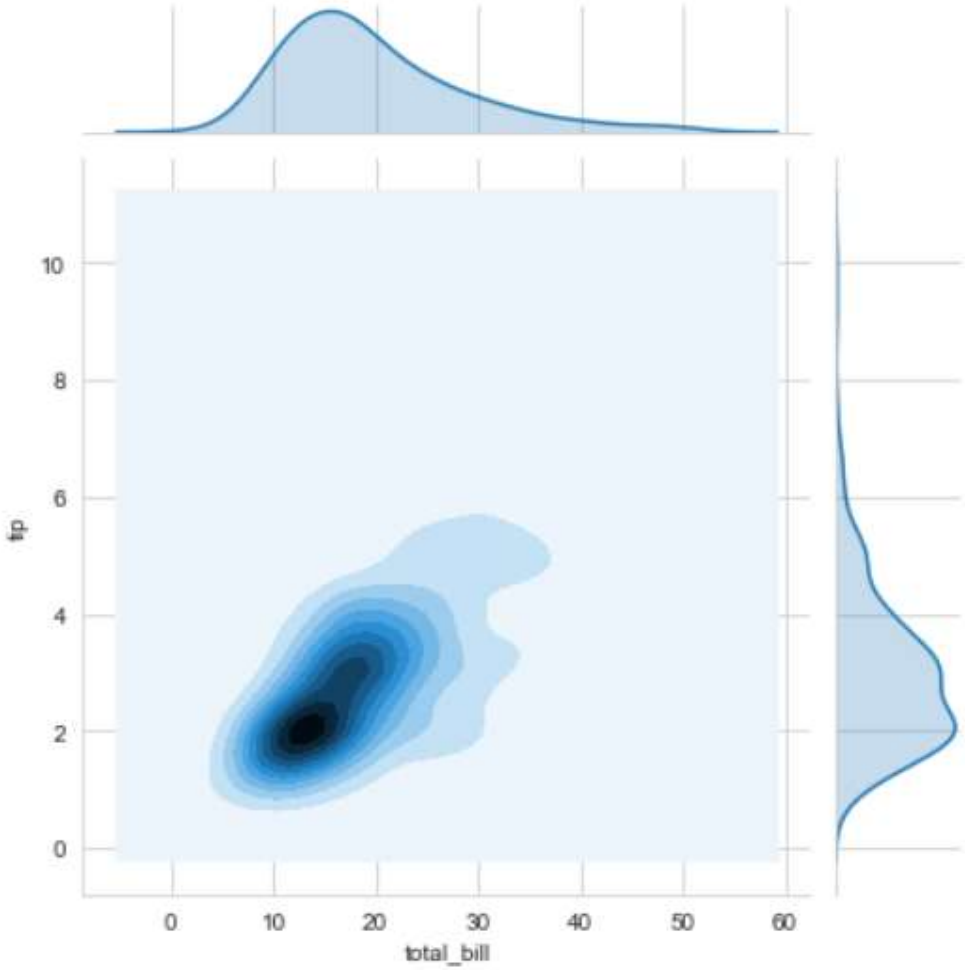
```
sns.jointplot(x ='total_bill', y ='tip', data = df)
```

**Output:**




```
sns.jointplot(x ='total_bill', y ='tip', data = df, kind ='kde')
```

# KDE shows the density where the points match up the most




**Explanation:**

- kind is a variable that helps us play around with the fact as to how do you want to visualise the data.It helps to see whats going inside the joinplot. The default is scatter and can be hex, reg(regression) or kde.
- x and y are two strings that are the column names and the data that column contains is used by specifying the data parameter.
- here we can see tips on the y axis and total bill on the x axis as well as a linear relationship between the two that suggests that the total bill increases with the tips.

**Pairplot**

It represents pairwise relation across the entire dataframe and supports an additional argument called **hue** for categorical separation. What it does basically is create a jointplot between every possible numerical column and takes a while if the dataframe is really huge.

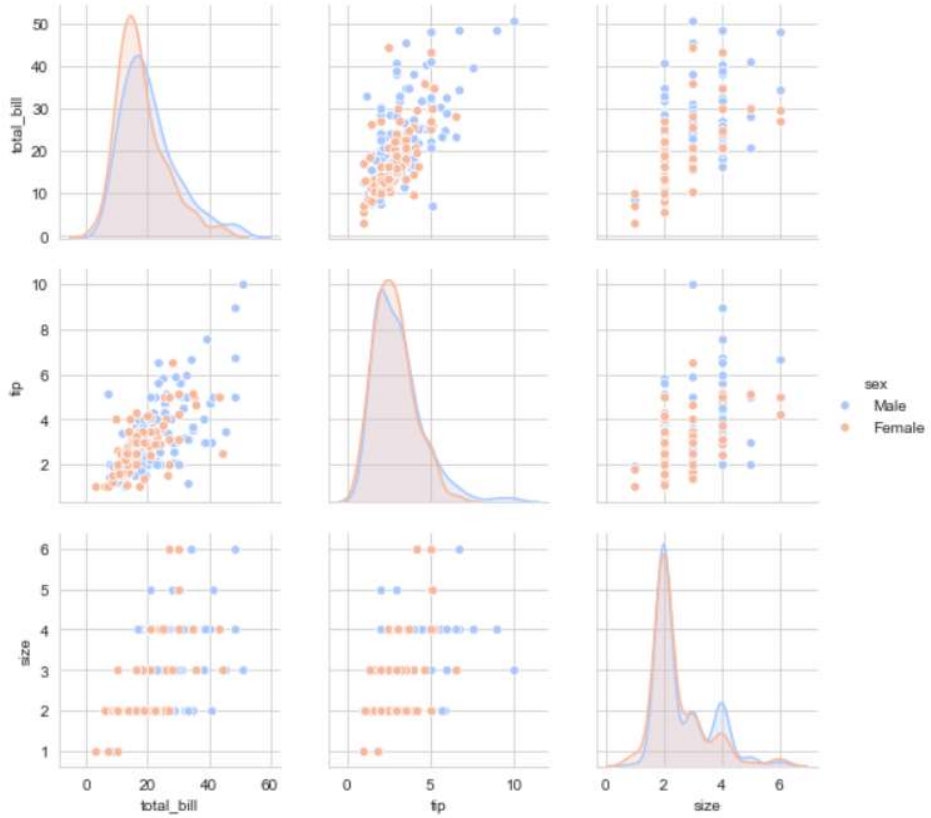
**Syntax:**

pairplot(data[, hue, hue_order, palette, …])

**Example:**

```
sns.pairplot(df, hue ="sex", palette ='coolwarm')
```

**Output:**




**Explanation:**

- hue sets up the categorical separation between the entries if the dataset.
- palette is used for designing the plots.

**Rugplot**

It plots datapoints in an array as sticks on an axis.Just like a distplot it takes a single column. Instead of drawing a histogram it creates dashes all across the plot. If you compare it with the joinplot you can see that what a jointplot does is that it counts the dashes and shows it as bins.

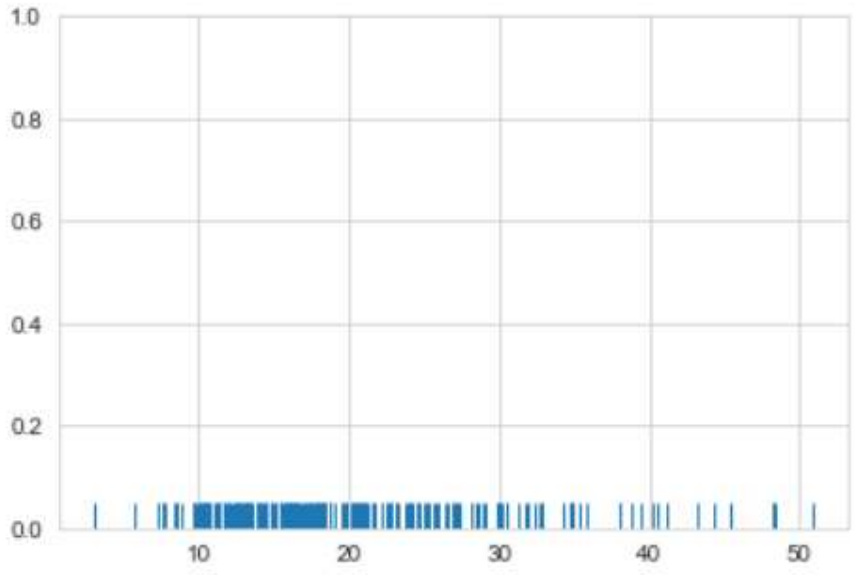
**Syntax:**

rugplot(a[, height, axis, ax])

**Example:**

```
sns.rugplot(df['total_bill'])
```

**Output:**




**Matrix plots in Seaborn**

Seaborn is a wonderful visualization library provided by python. It has several kinds of plots through which it provides the amazing visualization capabilities. Some of them include count plot, scatter plot, pair plots, regression plots, matrix plots and much more

**Heatmaps**

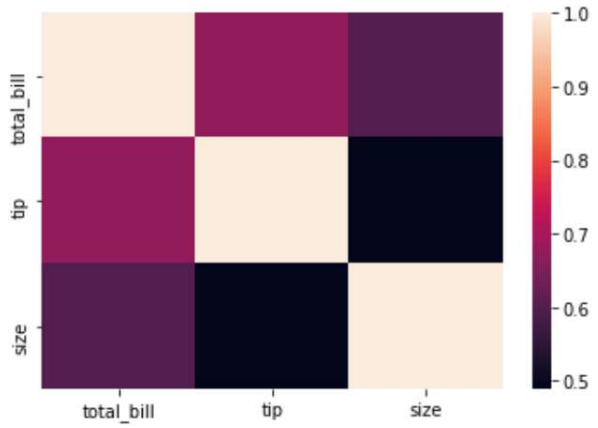
Heatmap is a way to show some sort of matrix plot. To use a heatmap the data should be in a matrix form. By matrix we mean that the index name and the column name must match in some way so that the data that we fill inside the cells are relevant.

```
import seaborn as sns
import matplotlib.pyplot as plt % matplotlib inline

dataset = sns.load_dataset('tips')
 dataset.head()

# correlation between the different parameters
tc = dataset.corr()

# plot a heatmap of the correlated data
sns.heatmap(tc)
```

| | total_bill | tip | size |
|---|---|---|---|
| **total_bill** | 1.000000 | 0.675734 | 0.598315 |
| **tip** | 0.675734 | 1.000000 | 0.489299 |
| **size** | 0.598315 | 0.489299 | 1.000000 |

**The correlation matrix**

```
<matplotlib.axes._subplots.AxesSubplot at 0x218f4177748>
```



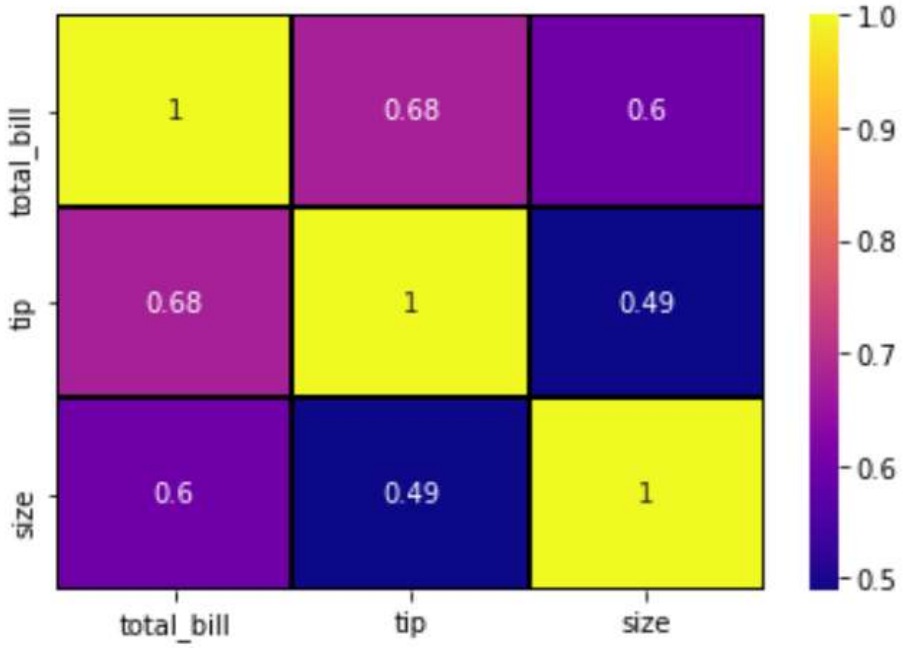

**Heatmap of the correlated matrix**

Inorder to obtain a better visualization with the heatmap, we can add the parameters such as annot, linewidth and line colour.

```
# import the necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt % matplotlib inline

# load the tips dataset
dataset = sns.load_dataset('tips')

# first five entries of the tips dataset
dataset.head()

# correlation between the different parameters
tc = dataset.corr()
sns.heatmap(tc, annot = True, cmap ='plasma', linecolor ='black', linewidths = 1)
```

Here is a plot that shows those attributes.

```
<matplotlib.axes._subplots.AxesSubplot at 0x218f413d4a8>
```




**Cluster maps**

Cluster maps use hierarchical clustering. It performs the clustering based on the similarity of the rows and columns.

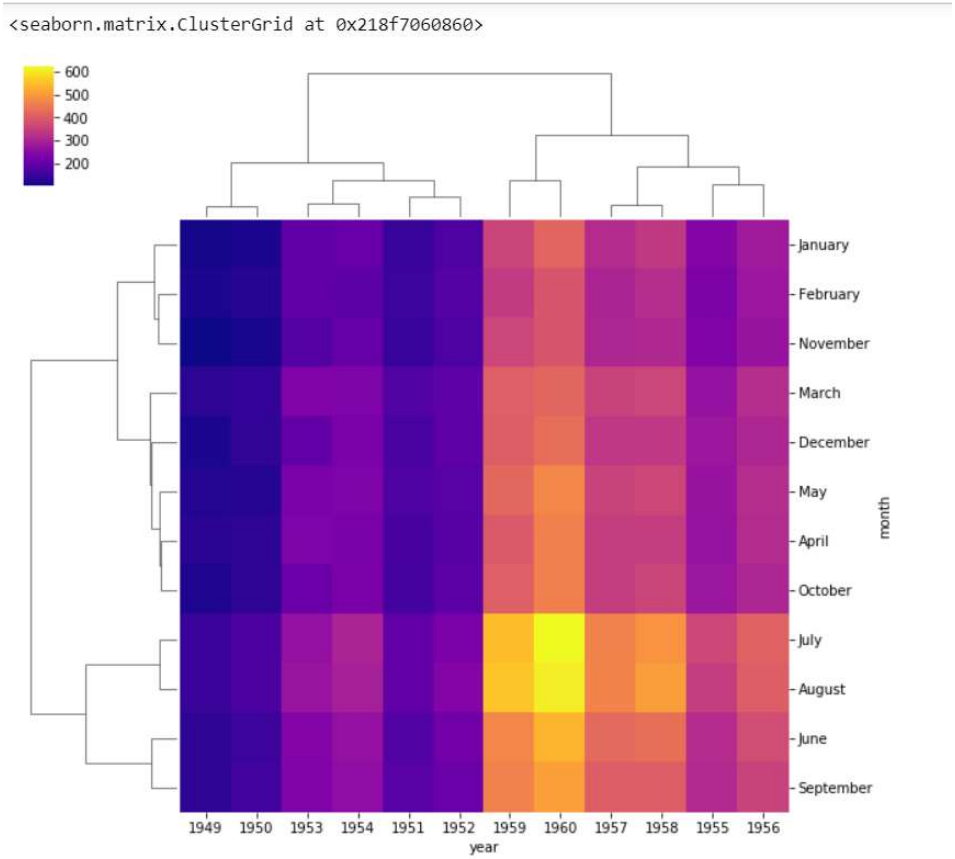
```
# import the necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt % matplotlib inline

# load the flights dataset
fd = sns.load_dataset('flights')

# make a dataframe of the data
df = pd.pivot_table(values ='passengers', index ='month',columns ='year', data = fd)

# first five entries of the dataset
df.head()

# make a clustermap from the dataset
sns.clustermap(df, cmap ='plasma')
```

| | year | month | passengers |
|---|---|---|---|
| 0 | 1949 | January | 112 |
| 1 | 1949 | February | 118 |
| 2 | 1949 | March | 132 |
| 3 | 1949 | April | 129 |
| 4 | 1949 | May | 121 |

**The first five entries of the dataset**

| year | 1949 | 1950 | 1951 | 1952 | 1953 | 1954 | 1955 | 1956 | 1957 | 1958 | 1959 | 1960 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| month | | | | | | | | | | | | |
| January | 112 | 115 | 145 | 171 | 196 | 204 | 242 | 284 | 315 | 340 | 360 | 417 |
| February | 118 | 126 | 150 | 180 | 196 | 188 | 233 | 277 | 301 | 318 | 342 | 391 |
| March | 132 | 141 | 178 | 193 | 236 | 235 | 267 | 317 | 356 | 362 | 406 | 419 |
| April | 129 | 135 | 163 | 181 | 235 | 227 | 269 | 313 | 348 | 348 | 396 | 461 |
| May | 121 | 125 | 172 | 183 | 229 | 234 | 270 | 318 | 355 | 363 | 420 | 472 |

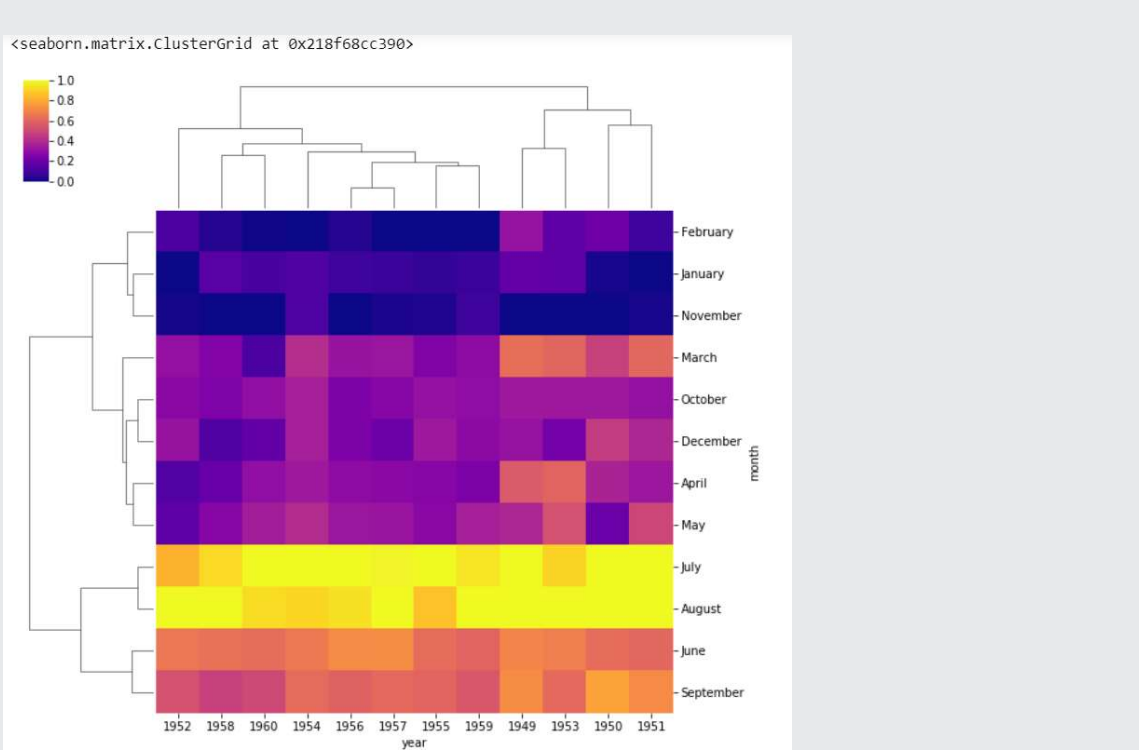
**The matrix created using the pivot table(first five entries)**




**Clustermap from the given data**

We can also change the scale of the color bar by using the standard_scale parameter.

```
# import the necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt % matplotlib inline

# load the flights dataset
fd = sns.load_dataset('flights')

# make a dataframe of the data
df = pd.pivot_table(values ='passengers',  index ='month', columns ='year', data = fd)

df.head()

# make a clustermap from the dataset
sns.clustermap(df, cmap ='plasma', standard_scale = 1)
```



**Regression Plots**

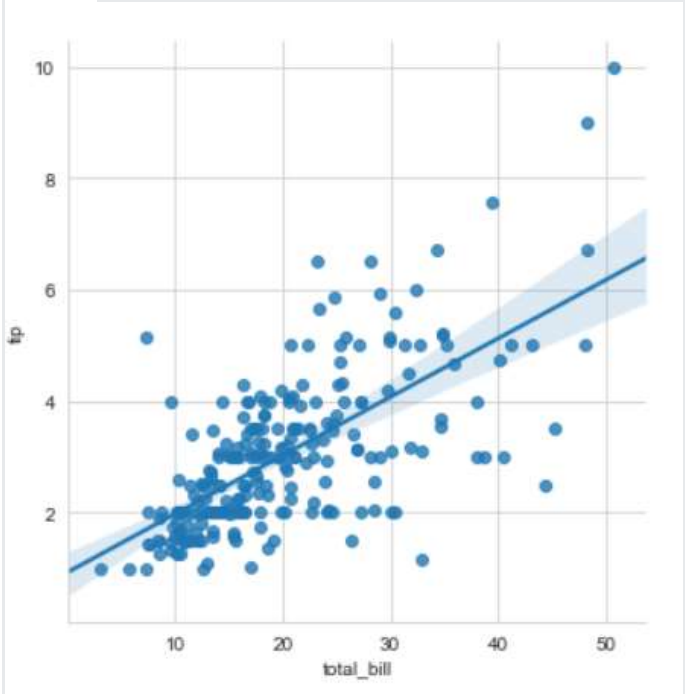
The regression plots in seaborn are primarily intended to add a visual guide that helps to emphasize patterns in a dataset during exploratory data analyses. Regression plots as the name suggests creates a regression line between 2 parameters and helps to visualize their linear relationships. This article deals with those kinds of plots in seaborn and shows the ways that can be adapted to change the size, aspect, ratio etc. of such plots.

Regression plots in seaborn can be easily implemented with the help of the lmplot() function. lmplot() can be understood as a function that basically creates a linear model plot. lmplot() makes a very simple linear regression plot.It creates a scatter plot with a linear fit on top of it.

**Simple linear plot**

```
import seaborn as sns

# load the dataset
dataset = sns.load_dataset('tips')

# the first five entries of the dataset
dataset.head()
sns.set_style('whitegrid')
sns.lmplot(x ='total_bill', y ='tip', data = dataset)
```

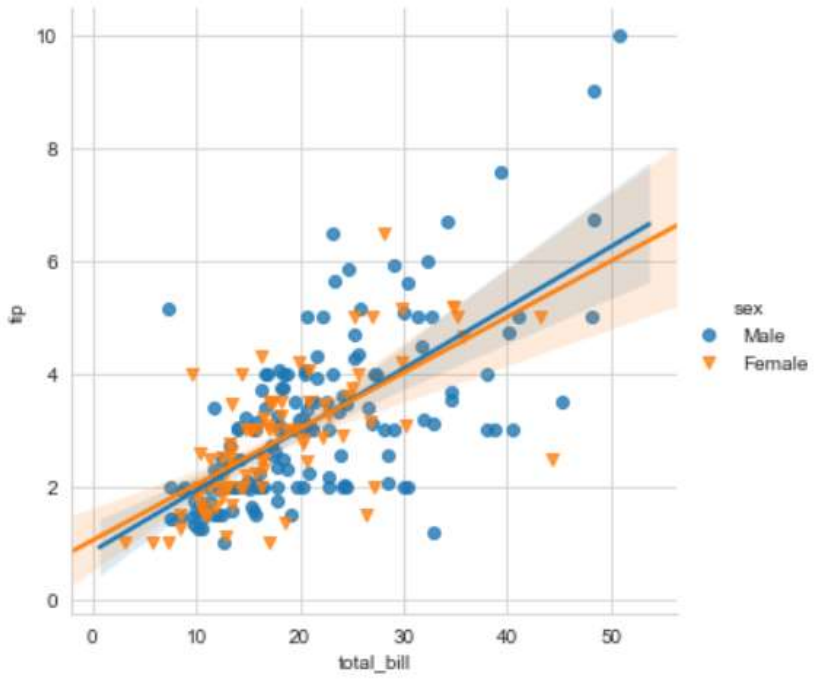
**Output**



**Linear plot with additional parameters**

```
sns.set_style('whitegrid')

sns.lmplot(x ='total_bill', y ='tip', data = dataset,

        hue ='sex', markers =['o', 'v'])
```

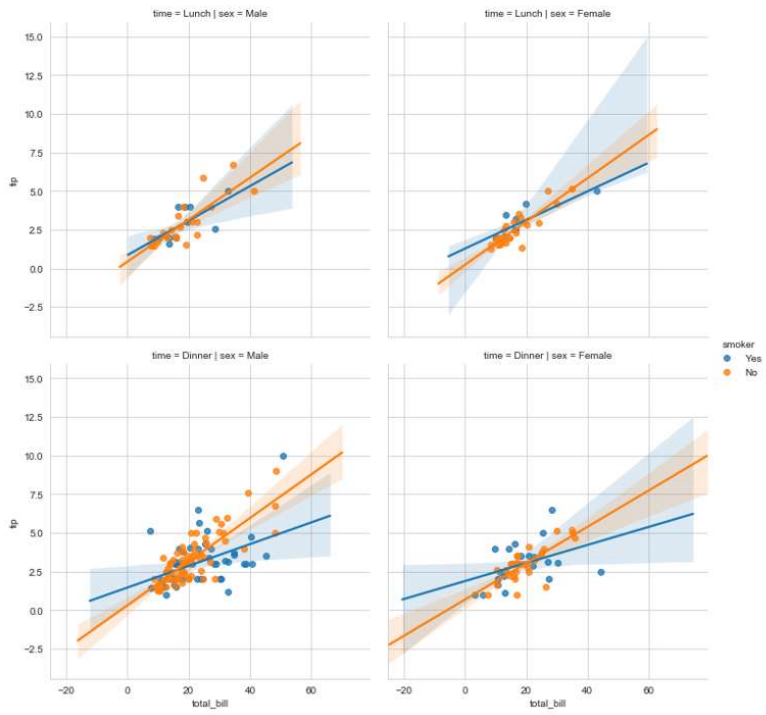
**Output**



**Displaying multiple plots**

```
sns.lmplot(x ='total_bill', y ='tip', data = dataset,
        col ='sex', row ='time', hue ='smoker')
```

**Output**



## Scikit-Learn (Sklearn)

Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python. This library, which is largely written in Python, is built upon NumPy, SciPy and Matplotlib.

### Installation

If you already installed NumPy and Scipy, following are the two easiest ways to install scikit-learn −

### Using pip

Following command can be used to install scikit-learn via pip −

pip install -U scikit-learn

### Features

Rather than focusing on loading, manipulating and summarising data, Scikit-learn library is focused on modeling the data. Some of the most popular groups of models provided by Sklearn are as follows −

Supervised Learning algorithms − Almost all the popular supervised learning algorithms, like Linear Regression, Support Vector Machine (SVM), Decision Tree etc., are the part of scikit-learn.

Unsupervised Learning algorithms − On the other hand, it also has all the popular unsupervised learning algorithms from clustering, factor analysis, PCA (Principal Component Analysis) to unsupervised neural networks.

Clustering − This model is used for grouping unlabeled data.

Cross Validation − It is used to check the accuracy of supervised models on unseen data.

Dimensionality Reduction − It is used for reducing the number of attributes in data which can be further used for summarisation, visualisation and feature selection.

Ensemble methods − As name suggest, it is used for combining the predictions of multiple supervised models.

Feature extraction − It is used to extract the features from data to define the attributes in image and text data.

Feature selection − It is used to identify useful attributes to create supervised models.

Open Source − It is open source library and also commercially usable under BSD license.

# Plotly-Python

The Plotly Python library is an interactive open-source library. This can be a very helpful tool for data visualization and understanding the data simply and easily. plotly graph objects are a high-level interface to plotly which are easy to use. It can plot various types of graphs and charts like scatter plots, line charts, bar charts, box plots, histograms, pie charts, etc.
Installation:

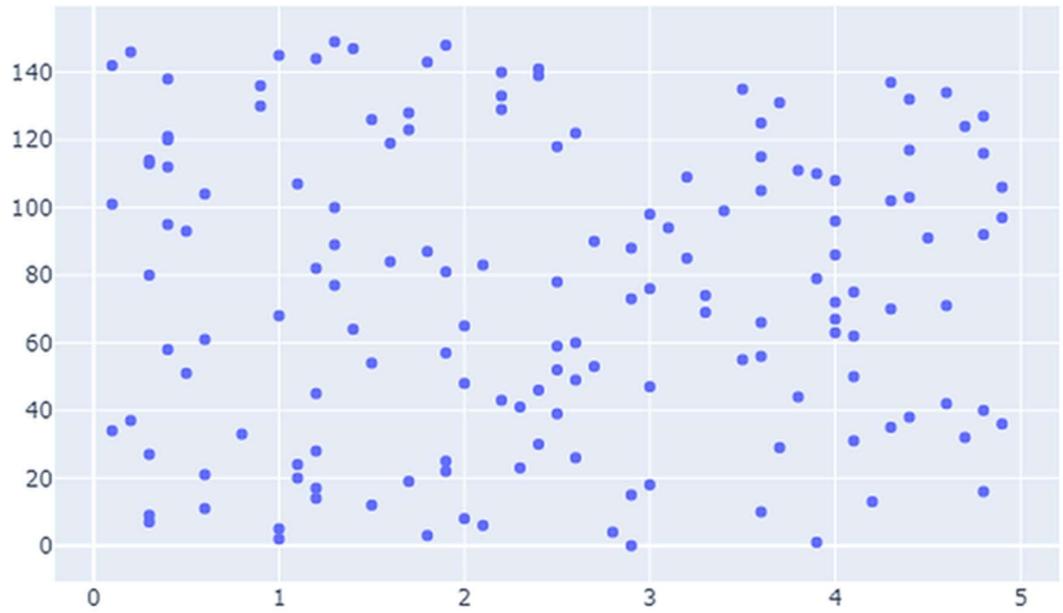
To install this module type the below command in the terminal.

**pip install plotly**

- **Scatter Plot:** Scatter plot represent values for two different numeric variables. They are mainly used for the representation of the relationship between two variables.

```
# import all required libraries
importnumpy as np
importplotly
importplotly.graph_objects as go
importplotly.offline as pyo
fromplotly.offline importinit_notebook_mode

init_notebook_mode(connected=True)

# generating 150 random integers
# from 1 to 50
x =np.random.randint(low=1, high=50, size=150)*0.1

# generating 150 random integers
# from 1 to 50
y =np.random.randint(low=1, high=50, size=150)*0.1

# plotting scatter plot
fig =go.Figure(data=go.Scatter(x=x, y=y, mode='markers'))

  fig.show()
```

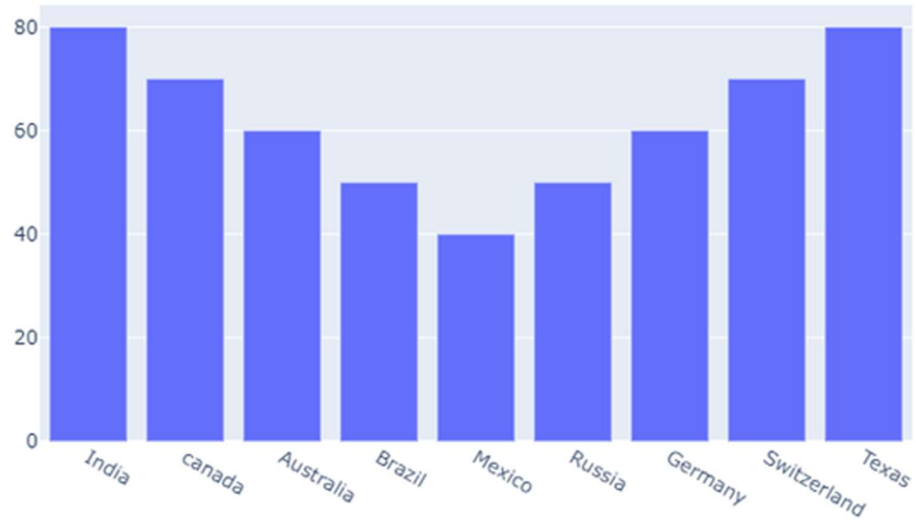
Output:




- **Bar charts:** Bar charts are used when we want to compare different groups of data and make inferences of which groups are highest and which groups are common and compare how one group is performing compared to others

```
# import all required libraries
importnumpy as np
importplotly
importplotly.graph_objects as go
importplotly.offline as pyo
fromplotly.offline importinit_notebook_mode

init_notebook_mode(connected =True)

# countries on x-axis
countries=['India', 'canada',
        'Australia','Brazil',
        'Mexico','Russia',
        'Germany','Switzerland',
        'Texas']

# plotting corresponding y for each
# country in x
fig =go.Figure([go.Bar(x=countries,
                y=[80,70,60,50,
                  40,50,60,70,80])])

  fig.show()
```

Output:

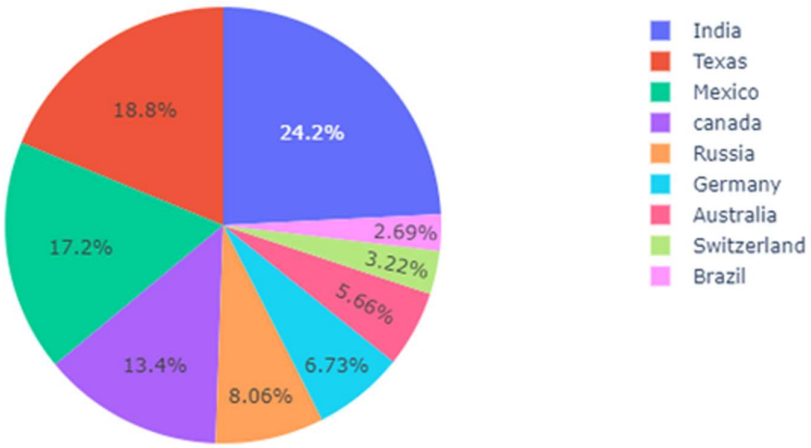



- **Pie chart:** A pie chart represents the distribution of different variables among total. In the pie chart each slice shows its contribution to the total amount.

```
# import all required libraries
importnumpy as np
importplotly
importplotly.graph_objects as go
importplotly.offline as pyo
fromplotly.offline importinit_notebook_mode

init_notebook_mode(connected =True)

# different individual parts in
# total chart
countries=['India', 'canada',
        'Australia','Brazil',
        'Mexico','Russia',
        'Germany','Switzerland',
        'Texas']

# values corresponding to each
# individual country present in
# countries
values =[4500, 2500, 1053, 500,
       3200, 1500, 1253, 600, 3500]
```

```
# plotting pie chart
fig =go.Figure(data=[go.Pie(labels=countries,
                values=values)])

  fig.show()
```

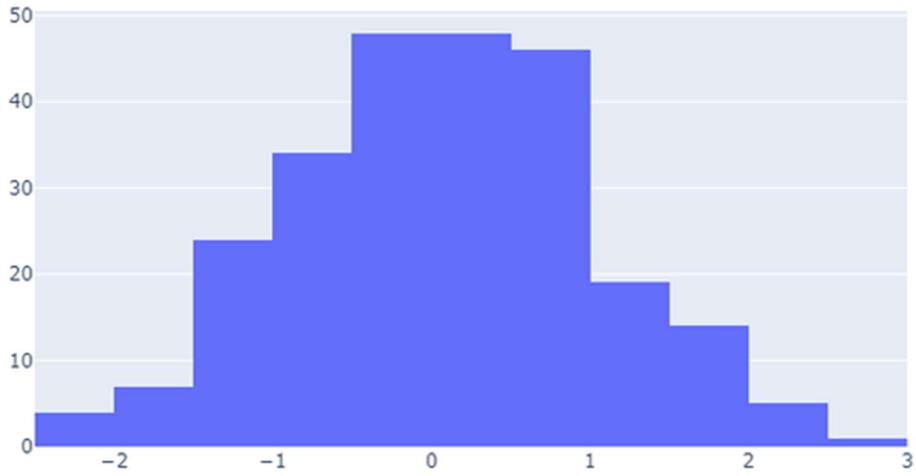
Output:




- **Histogram:** A histogram plots the continuous distribution of variable as series of bars and each bar indicates the frequency of the occurring value in a variable. In order to use a histogram, we simply require a variable that takes continuous numeric values

```
# import all required libraries
importnumpy as np
importplotly
importplotly.graph_objects as go
importplotly.offline as pyo
fromplotly.offline importinit_notebook_mode

init_notebook_mode(connected =True)

# save the state of random
np.random.seed(42)

# generating 250 random numbers
x =np.random.randn(250)

# plotting histogram for x
fig =go.Figure(data=[go.Histogram(x=x)])
```

fig.show()
Output:




- **Box plot:** A box plot is the representation of a statistical summary. Minimum, First Quartile, Median, Third Quartile, Maximum.

```
# import all required libraries
importnumpy as np
importplotly
importplotly.graph_objects as go
importplotly.offline as pyo
fromplotly.offline importinit_notebook_mode

init_notebook_mode(connected =True)

np.random.seed(42)

# generating 50 random numbers
y =np.random.randn(50)

# generating 50 random numbers
y1 =np.random.randn(50)
fig =go.Figure()

# updating the figure with y
fig.add_trace(go.Box(y=y))
```

```
# updating the figure with y1
fig.add_trace(go.Box(y=y1))

  fig.show()
```

Output: