**Panda's basics for data manipulation in Python**

**Introduction to Data Manipulation**

Data manipulation is the process of altering or transforming data to extract, organize, modify, or analyze it in order to derive meaningful insights or meet specific requirements. We can perform various operations on the data like as filtering, sorting, aggregating, joining, merging, and transforming etc. Data manipulation is commonly performed in data analysis, database management, and programming contexts. There are following common techniques and operations involved in data manipulation:

A. **Filtering**: Selecting a subset of data from the database based on specified required criteria. For example, selecting all sales records where the purchase amount is greater than a certain value.
B. **Sorting:** Arranging the data in a specific order based on one or more attributes. For instance, sorting a list of customer names in alphabetical order.
C. **Aggregating**: Combining multiple data records to create summary statistics or metrics. This can include operations like calculating the total sales, average price, or maximum value.
D. **Joining/Merging**: Combining data from multiple sources based on a common attribute or key. This is often used to consolidate information from different tables or datasets.
E. **Transforming:** Modifying the structure or format of data. This can involve operations like converting data types, renaming columns, or splitting and combining values.
F. **Cleaning:** Correcting or removing errors, inconsistencies, or missing values in the data. This step ensures data quality and reliability.
G. **Grouping:** Grouping data based on specific criteria to analyze subsets of data. For instance, grouping sales data by region or product category.
H. **Summarizing:** Generating descriptive statistics or summarizing data based on certain criteria. This includes operations like calculating the mean, median, or standard deviation.

   Data manipulation can be performed using various tools and technologies, like spreadsheet software (e.g., Microsoft Excel, Google Sheets), programming languages (e.g., Python, R, SQL), data manipulation libraries (e.g., pandas in Python), and database management systems (e.g., MySQL, PostgreSQL).

There are following 4 queries that can be used for the data manipulation in SQL:

- **SELECT** – to query data in the database.
- I**NSERT** – to insert data into a table.
- **UPDATE** – to update data in a table.
- **DELETE** – to delete data from a table.

## 5.1 Introduction to NumPy

NumPy stands for the Numerical python. It is a python package for the computation and processing of the multidimensional and single dimensional array elements. **Travis Oliphant** created NumPy package in 2005.

- It is an open-source project and you can use it freely. NumPy offers various powerful data structures, implementing multi-dimensional arrays and matrices.
- NumPy is written partially in Python, but most of the parts that require fast computation are written in C or C++.
- NumPy is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
- NumPy provides a convenient and efficient way to handle the vast amount of data.

## 5.2 NumPy Package in Python

NumPy doesn't come bundled with Python. We have to install it using the **python pip installer.**

**$ pip install NumPy**

After the installation of the NumPy Package we can check whether package is installed or not y using the **import** keyword:

```
Import  numpy          # use of NumPy library
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

- **Checking NumPy Version:** The version string is stored under __version__ attribute.
  ```
  import numpy as np
  print (np.__version__)
  ```

## 5.3 Importing NumPy

NumPy is imported under the **np** alias. We can use the following **syntax:**

**Import NumPy as np** **# np** means **NumPy package**

**For Example:**

```
import  numpy as np       # import NumPy library
arr = np.array([10, 20, 30, 40, 50])
print (arr)
print (type(arr))
```

## 5.4 Creating different arrays using NumPy

We can create different kinds of array as mentioned below:

- **0-D Array:** It is a special type of array that contains single elements in an array.

  ```
  import numpy as np
  ```

```
arr = np.array (42)                    # Zero – D Array
print (arr)
```

- **1-D Array**

  1-D arrays in numpy are one dimension that can be thought of a list where you can access the elements with the help of indexing.

  **Syntax: array_name = np. Array ([item1, item2, item3...])**

  **For Example:**

```
import numpy as np
arr1 = np. array ([10, 20, 30, 40, 50])
print (arr1)
```

- **2-D Array**

  2-D arrays in numpy are two dimensions array that can be distinguished based on the number of square brackets used. NumPy has a whole sub module dedicated towards matrix operations called (**numpy.mat).**

  **Syntax: array_name = np. Array ([item1, item2, item3],[elem1,elem2,elem3])**

```
import numpy as np
my_arr = np.array ([[25, 22, 29], [14, 15, 16]])    # 2-D Array
print (my_arr)
```

- **3-D Array**

  The 3-D arrays in numpy are the three-dimension array that can have three square brackets.

  **Syntax: my_arr = np.array([[[iem1, item2, item3], [obj1, obj2, obj3]], [[elem1, elem2, elem3]]])**

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

- **Multi-Dimensional Array**

  We can create multidimensional array by using the **ndim** argument.

  **Syntax: arr-name = np. array ([1, 2, 3, 4], ndmin = value)**

  **For Example:**

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

## 5.5 Array functions and Methods

There is a vast range of built-in operations that we can perform on these arrays.

1. Ndim – It returns the dimensions of the array.
2. Itemsize – It calculates the byte size of each element.
3. Dtype – It can determine the data type of the element.
4. Reshape – It provides a new view.
5. Slicing – It extracts a particular set of elements.
6. Linspace – Returns evenly spaced elements.
7. max/min, sum, sqrt
8. Ravel – It converts the array into a single line.

## 5.6 Different Matrix Operations

Python has powerful features that offers matrix features. The matrix can be implemented by using list or array in the python.

There are various operations can be performed on the matrix as below:

**1. Add ():**- This function is used to perform element wise matrix addition.

2. **Subtract ():-** This function is used to perform element wise matrix subtraction.

3. **Divide ():-** This function is used to perform element wise matrix division.

4. **Multiply ():-** This function is used to perform element wise matrix multiplication.

**5. Dot ():-** This function is used to compute the matrix multiplication, rather than element wise multiplication.

**6. Sqrt ():-** This function is used to compute the square root of each element of matrix.

**7. Sum(x, axis):-** This function is used to add all the elements in matrix. Optional "axis" argument computes the column sum if axis is 0 and row sum if axis is 1.

8. **"T" (Transpose):-** This argument is used to transpose the specified matrix.

```
Import NumPy
x = numpy.array([[6, 12], [8, 13]])  # initializing matrices
y = numpy.array([[7, 14], [9, 15]])
print ("The element wise addition of matrix is : ")
print (numpy.add(x,y))          # using add() to add matrices
print ("The element wise subtraction of matrix is : ")
print (numpy.subtract(x,y))     # using subtract() to subtract matrices
print ("The element wise division of matrix is : ")
print (numpy.divide(x,y))       # using divide() to divide matrices
print ("The element wise multiplication of matrix is : ")
print (numpy.multiply(x,y))     # using multiply() to multiply matrices element wise
print ("The product of matrices is : ")
print (numpy.dot(x,y))          # using dot() to multiply matrices
print ("The element wise square root is : ")
print (numpy.sqrt(x))           # using sqrt() to print the square root of matrix
```

```
print ("The summation of all matrix element is : ")
print (numpy.sum(y)) #using sum() to print summation of all elements of matrix
 print ("The column wise summation of all matrix  is : ")
print (numpy.sum(y,axis=0)) # using sum(axis=0) Sum of all columns of matrix
print ("The row wise summation of all matrix  is : ")
print (numpy.sum(y,axis=1))  # using sum(axis=1) Sum of all columns of matrix
print ("The transpose of given matrix is : ")
print (y.T)        # using "T" to transpose the matrix
```

**Output:**

```
The element wise addition of matrix is :
[[13 26]
 [17 28]]
The element wise subtraction of matrix is :
[[-1 -2]
 [-1 -2]]
The element wise division of matrix is :
[[0.85714286 0.85714286]
 [0.88888889 0.86666667]]
The element wise multiplication of matrix is :
[[ 42 168]
 [ 72 195]]
The product of matrices is :
[[150 264]
 [173 307]]
The element wise square root is :
[[2.44948974 3.46410162]
 [2.82842712 3.60555128]]
The summation of all matrix element is :
45
The column wise summation of all matrix  is :
[16 29]
The row wise summation of all matrix  is :
[21 24]
The transpose of given matrix is :
[[ 7  9]
 [14 15]]
```

## 5.8 Random Numbers

The random is a module present in the NumPy library. This module contains the functions which are used for generating random numbers. This module contains some simple random data generation methods, permutation & distribution functions, and random generator functions.

### A. Simple random data

This function of random module is used to generate random numbers or values in a given range.

```
import   numpy as np
a=  np.random.rand(3,3)
print(a)
```

**Output:**

```
[[0.20514791 0.83531785 0.24420771]
 [0.15143734 0.57222014 0.96365475]
 [0.48860394 0.29093862 0.13224575]]
```

## 5.9 Generate Numbers between a range

Python provides a function named randrange() in the random package that can produce random numbers from a given range . There are various method that can be used for generating random numbers between a ranges:

**Method 1: Generate random integers using random.randrange() method**

```
import random
print("Random integers between 0 and 8: ")
for i in range(7, 13):
    y = random.randrange(8)    # use of randrange method
    print(y)
```

**Output:**

```
Random integers between 0 and 8:
0
7
7
6
0
2
```

**Method 2: Generate random integers using random.uniform() method**

The method, “random.uniform()” is defined in the “random” module. It Returns the generated floating-point random number between the lower limit and upper limit.

```
import random
 print("Random integers between 0 and 9: ")
for i in range(4, 11):
    y = random.uniform(4, 10)
    print(y)
```

```
Random integers between 0 and 9:
4.691959269925425
4.897539394411522
8.478985024384922
4.849238164477233
6.308700725404181
8.135630251209456
5.967500590118932
```

**Method 3: Generate random integers using randbelow() method**

This method is used for handling crucial information including cryptographically secure passwords, account authentication, security tokens, and related secre e secrets module is utilized to generate random integers. We can use randbelow( tion from the **secrets module** to generate random integers.

```
5
8
7
7
7
5
```

```
from secrets import randbelow
 for _ in range(3, 9):
    print(randbelow(10))  # randbelow method
```

**Method 4: Generate random integers using the random.randint() method**

Python provides a random module to generate random numbers. To generate random numbers we have used the random function along with the use of the (random.randint) function. randint accepts two parameters, a **starting point**, and an **ending point**. Both should be integers and the first value should always be less than the second.

```
import numpy as np
def Rand(start, end, num):
   res = []
```

```
    for j in range(num):
        res.append(np.random.randint(start, end))
    return res
num = 10
start = 20
end = 40
print(Rand(start, end, num))
```

**Output**

**Pandas basics for data manipulation in Python**

## 1. Understanding the Pandas Library

Python Pandas is defined as an open-source library that provides high-performance data manipulation in Python. It is used for data analysis in Python and developed by Wes McKinney in 2008.

Pandas is built on top of the Numpy package, means Numpy is required for operating the Pandas. The data produced by Pandas are often used as input for plotting functions of Matplotlib, statistical analysis in SciPy, and machine learning algorithms in Scikit-learn.

**Key Features of Pandas**

1. **Handling of huge volume of data**

   The Pandas library offers a fast and efficient way to manage and explore data. It providing us key feature like **Series** and **DataFrames** that help us data manipulation and representation of data in various ways.

2. **Alignment and indexing**

   Labelling and Indexing are the key features that give a new formatting to our data. If we have data but we don't know where it belongs and what it tells us about. So we can use label features for labelling of data as per requirement.

3. **Handling missing data**

   As in real time, Data is very crude in nature. There are various source of data through which data is generated in real time. So there may be some missing data or value. Therefore, it is pertinent to handle the missing values properly so that they do not adulterate our results.

4. **Cleaning up data**

   It is true, Data can be very crude. So we may have the really messy data, so performing any analysis over such data would lead to severely wrong results. Thus it is very important that we clean our data up.

5. **Input and output tools**

   Pandas provide built-in tools for the purpose of reading and writing data. While analysing data, we need to read and write data into data structures, web service, databases, etc.

6. **Multiple file formats supported**

   In real time, Data is generating via various platform so we can be found in many different file formats. So pandas has the key libraries that can be used for data analysis and can read various file formats like JSON or CSV, Excel etc.

7. **Merging and joining of datasets**

   Data analysis is very key aspects in real world. While analysing data, we constantly need to merge and join multiple datasets to create a final dataset. This is important because if the datasets aren't merged or joined properly, then it is going to affect the results. Pandas can help to merge various datasets, with extreme efficiency so that we don't face any problems while analysing the data.

8. **Multiple features for Time Series**

   These features include the likes of moving window statistics and frequency conversion.

9. **Optimized performance**

   Pandas is said to have a really optimized performance, which makes it really fast and suitable for data science. The critical code for Pandas is written in C or Cython, which makes it extremely responsive and fast.

10. **Visualization of data**

    Visualizing the data is an important part of data science. It is what make the results of the study understandable by human eyes. Pandas have an in-built ability to help you plot your data and see the various kinds of graphs formed.

11. **Grouping of data**

    Pandas also has the ability to separate your data and grouping it according to the criteria you want. With the help of the features of Pandas like GroupBy, you can split data into categories of your choice, according to the criteria you set. The GroupBy function splits the data, implements a function and then combines the results.

12. **Masking data**

    When in a case certain data is not needed for analysis and thus it is important that you filter your data according to the things you want from it. Using the mask function in Pandas allows you exactly to do that. It is extremely useful since whenever it finds data which meets the criteria you set for elimination, it turns the data into a missing value.

13. **Data is unique**

    Data always has a lot of repetition, therefore it is important that you are able to analyse data which has only unique values. By using the function dataset.column.unique (), we can eliminate the data repetition.

14. **Perform mathematical operations on the data**

    The built-in function in Pandas allows you to implement a mathematical operation on the data. Sometimes the dataset you have, is just not of the correct order. So by using the mathematical operations we can arrange that data as per our criteria.

Pandas offers three types of data structures:

| Description | Dimensions | Description |
|---|---|---|
| Series | 1 | 1D labelled homogeneous array, size immutable. |
| Data Frames | 2 | General 2D labelled, size-mutable tabular structure with potentially heterogeneously typed columns. |

## 1.1 Series in Pandas

Series is a one-dimensional array that is capable of storing various data types. The axis labels are collectively called indexes.

**(a)Creating a Series using list:**

```
Import pandas as pd
a = [8, 9, 5]                # list
mydata = pd.Series(a)        # creation of series
print(mydata)
```

**Output:**

```
0    8
1    9
2    5
dtype: int64
```

**(b)Creating series with specific labels:**

```
import pandas as pd
a = [58, 68, 72]
myvar = pd.Series (a, index = ["a", "b", "c"])
print(myvar)
```

**Output:**

```
a    58
b    68
c    72
dtype: int64
```

(c) **Creating a series by using dictionary**

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series (calories)
print (myvar)
```

**Output:**

```
day1    420
day2    380
day3    390
dtype: int64
```

## 1.2 Data frame and operations

A Data frame is a two-dimensional data structure. Data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

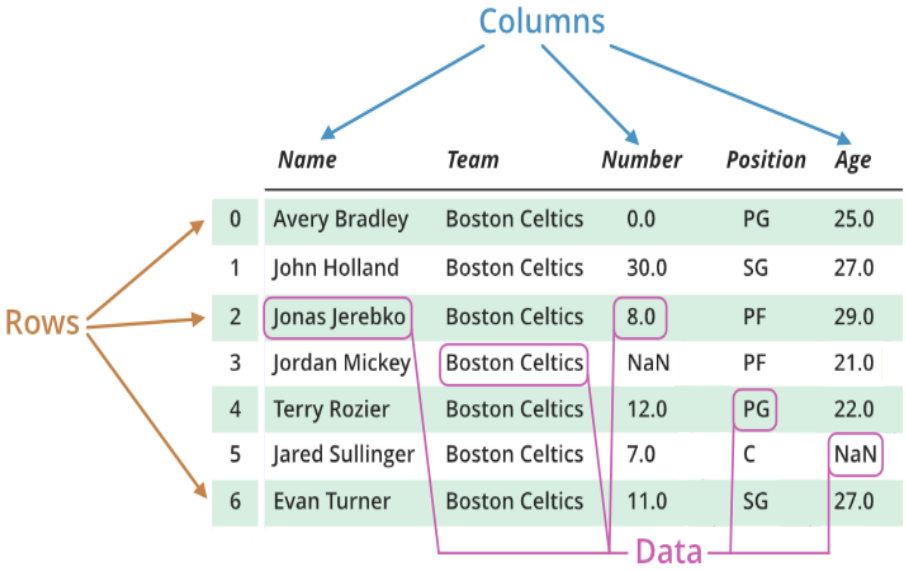
**Creating Data Frame**

Pandas DataFrame will be created by loading the datasets from existing SQL Database, CSV file, or an Excel file. DataFrame also can be created from lists, dictionaries etc. Data frame can be created by using the **DataFrame ( )** constructor.

```
import pandas as pd
df = pd.DataFrame()          # Calling DataFrame constructor
print (df)
data = ['BCA', 'MCA', 'BTech', 'MTech', 'MBA', 'PHD', 'Bsc']    # list of strings
df = pd.DataFrame(data)      # Calling DataFrame constructor on list
print (df)
```

**Output**

```
      0
0   BCA
1   MCA
2  BTech
3  MTech
4   MBA
5   PHD
6   Bsc
```




| | Name | Team | Number | Position | Age |
|---|---|---|---|---|---|
| 0 | Avery Bradley | Boston Celtics | 0.0 | PG | 25.0 |
| 1 | John Holland | Boston Celtics | 30.0 | SG | 27.0 |
| 2 | Jonas Jerebko | Boston Celtics | 8.0 | PF | 29.0 |
| 3 | Jordan Mickey | Boston Celtics | NaN | PF | 21.0 |
| 4 | Terry Rozier | Boston Celtics | 12.0 | PG | 22.0 |
| 5 | Jared Sullinger | Boston Celtics | 7.0 | C | NaN |
| 6 | Evan Turner | Boston Celtics | 11.0 | SG | 27.0 |

## 2. Creating Series using Pandas

**Series ( )** function is used to create a series in Pandas Library. We can create a series in different ways as below:

**Creating an empty Series:**

```
Import pandas as pd
ser = pd.Series()
print (ser)                    # By default, the data type of Series is float.
```

### 2.1 Different Series Attributes

We can Series by using the use different inputs like Numpy Array, Dict, and List etc. The Series attribute is used to define any information related to the Series object such as size, datatype. etc. Below are some of the attributes that we can use to get the information about the Series object:

| Attributes | Description |
|---|---|
| **Series.index** | Defines the index of the Series. |
| **Series.shape** | It returns a tuple of shape of the data. |
| **Series.dtype** | It returns the data type of the data. |
| **Series.size** | It returns the size of the data. |
| **Series.empty** | It returns True if Series object is empty, otherwise returns false. |

| | |
|---|---|
| **Series.hasnans** | It returns True if there are any NaN values, otherwise returns false. |
| **Series.nbytes** | It returns the number of bytes in the data. |
| **Series.ndim** | It returns the number of dimensions in the data. |
| **Series.itemsize** | It returns the size of the datatype of item. |

A. **Creating Series using Numpy array:**

We have to import the numpy module and then use array () function in the program. If the data is ndarray, then the passed index must be of the same length.

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

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

B. **Create a Series from dictionary**

We can also create a Series from dictionary as mentioned below:

```
import pandas as pd
import numpy as np
info = {'x' : 0., 'y' : 1., 'z' : 2.}
a = pd.Series (info)
print (a)
```

```
x    0.0
y    1.0
z    2.0
dtype: float64
```

C. **Create a Series using Scalar:**

We can take the scalar values for creating a series but the index must be provided. The scalar value will be repeated for matching the length of the index.

```
import pandas as pd
import numpy as np
x = pd.Series (4, index= [0, 1, 2, 3])
print (x)
```

```
0    4
1    4
2    4
3    4
dtype: int64
```

D. **Accessing element of Series**

We can access the elements of the Series by using the following two methods:

- **Accessing Element from Series with Position**

  To access the element from a series, we need to use the index operator [ ]. The index must be an integer.

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

**Output**

```
2    A
3    C
4    D
5    E
6    L
dtype: object
```

```
# creating simple array
data = np.array(['C','D','A','C','D','E', 'L','H','I','J','A','S','O','L','A'])
ser1 = pd.Series (data)
print (ser1[2:7])    # for retrieve the elements, slice method used
```

- **Accessing Element Using Label (index)**

  In order to access an element from series, we have to set values by index label. A Series is like a fixed-size dictionary in that you can get and set values by index label.

```
Import pandas as pd
Import numpy as np
data = np.array(['C','D','A','C','D','L', 'H','I'])
ser = pd.Series(data, index=[1,2,3,4,5,6,7,8])
print (ser[6])
```

**Output:**

L

## 2.2 Series vs List

There are the key attributes of python list and Series:

| Python List | Python Series |
|---|---|
| A list is an ordered collection of elements, where each element has a specific index starting from 0. | Each element in the Series can be labelled with a unique index, which can be any hashable data type. |
| Lists are mutable, meaning you can add, remove, or modify elements after creating the list. | The size of a Series can be changed dynamically by adding or removing elements. |
| The elements in a list can be accessed by their index values. | Pandas Series supports vectorised operations, meaning mathematical operations can be performed on an entire Series. |
| The indexes of a list are always integers. | A series can store only similar kinds of data types to store the data. |
| A list can hold duplicate values. | Pandas Series can gracefully handle missing or NaN (Not a Number) values. |
| A list can store different types of data like string, int, float etc. | A Series can be sliced like a Numpy array using labels or integer positions. |
| | A scalar value can be broadcast to all the elements in the Series. |
| | Pandas Series can be created from a variety of data sources, including Numpy arrays, lists, and dictionaries. |

## 2.3 Series Operations

There are following operations that we can perform on the list:

| FUNCTION | DESCRIPTION |
|---|---|
| add() | Method is used to add series or list like objects with same length to the caller series |
| sub() | Method is used to subtract series or list like objects with same length from the caller series |
| mul() | Method is used to multiply series or list like objects with same length with the caller series |
| div() | Method is used to divide series or list like objects with same length by the caller series |
| sum() | Returns the sum of the values for the requested axis |
| prod() | Returns the product of the values for the requested axis |
| mean() | Returns the mean of the values for the requested axis |
| pow() | Method is used to put each element of passed series as exponential power of caller series and returned the results |
| abs() | Method is used to get the absolute numeric value of each element in Series/Data Frame |
| cov() | Method is used to find covariance of two series |

**A. add ( ):** Python Series. Add () is used to add series or list like objects with same length to the caller series.

**Syntax:** Series. add (other, level=None, fill_value=None, axis=0)

**Parameters:**

**Other:** other series or list type to be added into caller series

**fill_value:** Value to be replaced by NaN in series/list before adding

**Level:** integer value of level in case of multi index

**Return type:** Caller series with added values

**For Example:**

```
import pandas as pd
 # Reading csv file from source
data = pd.read_csv('C:/Users/Lenovo/OneDrive/Desktop/data.csv')
data
short_data = data.head ()                    # creating short data of 5 rows
list =[50, 80, 90, 125, 205]                 # creating list with 5 values
# creating new column
short_data["Added values"]= short_data["Salary"].add(list)
# display
print(short_data.to_string())    // to print whole data_frame
```

**B. Sub ( ):**

Python is a great language for doing data analysis, primarily because of the fantastic ecosystem of data-centric Python packages. Pandas is one of those packages and makes importing and analysing data much easier.

Python **Series. Sub ()** is used to subtract series or list like objects with same length from the caller series.

**Syntax:** Series. Sub (other, level=none, fill_value=none, axis=0)

**Parameters:**

- Other: other series or list type to be subtracted from caller series
- fill_value: Value to be replaced by NaN in series/list before subtracting
- Level: integer value of level in case of multi index
- Return type: Caller series with subtracted values

```
Import pandas as pd
# Reading csv file from source
data = pd.read_csv('C:/Users/Lenovo/OneDrive/Desktop/data.csv')
data
short_data = data.head ()              # creating short data of 5 rows
list =[50, 80, 90, 125, 205]           # creating list with 5 values
# adding list data
# creating new column
short_data ["Added values"] = short_data ["Salary"].sub (list)
 # Display
print (short_data.to_string ())
```

## 2.4 Series from CSV file

Pandas series is a One-dimensional ndarray with axis labels. The labels need not be unique but must be a hashable type.

We can use **Series.from_csv()** function to read a csv file into a series. It is preferable to use the more powerful pandas.read_csv() for most general purposes.

**Syntax:** Series.from_csv (path, sep=', ', parse_dates=True, header=None, index_col=0, encoding=None, infer_datetime_format=False)

**Parameter:**

**Path:** string file path or file handle / StringIO

**Sep:** Field delimiter

**parse_dates:** Parse dates. Different default from read_table

**Header:** Row to use as header (skip prior rows)

**index_col:** Column to use for index

**Encoding:** a string representing the encoding to use if the contents are non-ascii

**infer_datetime_format:** If True and parse_dates is **True** for a column, try to infer the date time format based on the first date time string.

**For Example:**

```
Import pandas as pd
 # Reading csv file from source
data = pd.read_csv('C:/Users/Lenovo/OneDrive/Desktop/data.csv')
data
short_data = data.head ()                    # creating short data of 5 rows
list = [50, 80, 90, 125, 205]                # creating list with 5 values
# creating new column
short_data ["Added values"] = short_data ["Salary"].add (list)
# Display
print (short_data.to_string())    // to print whole data_frame
```

## 2.5 Different functions in Series

There are various functions in series that can be used for the various purpose.

| FUNCTION | DESCRIPTION |
|---|---|
| Series() | A pandas Series can be created with the Series () constructor method. This constructor method accepts a variety of inputs |
| combine_first() | Method is used to combine two series into one |
| count() | Returns number of non-NA/null observations in the Series |
| size() | Returns the number of elements in the underlying data |
| name() | Method allows to give a name to a Series object, i.e. to the column |
| is_unique() | Method returns Boolean if values in the object are unique |
| idxmax() | Method to extract the index positions of the highest values in a Series |
| idxmin() | Method to extract the index positions of the lowest values in a Series |
| sort_values() | Method is called on a Series to sort the values in ascending or descending order |
| sort_index() | Method is called on a pandas Series to sort it by the index instead of its values |
| head() | Method is used to return a specified number of rows from the beginning of a Series. The method returns a brand new Series |
| tail() | Method is used to return a specified number of rows from the end of a Series. The method returns a brand new Series |
| le() | Used to compare every element of Caller series with passed series.It returns True for every element which is Less than or Equal to the element in passed series |

| | |
|---|---|
| ne() | Used to compare every element of Caller series with passed series. It returns True for every element which is Not Equal to the element in passed series |
| ge() | Used to compare every element of Caller series with passed series. It returns True for every element which is Greater than or Equal to the element in passed series |
| eq() | Used to compare every element of Caller series with passed series. It returns True for every element which is Equal to the element in passed series |
| gt() | Used to compare two series and return Boolean value for every respective element |
| lt() | Used to compare two series and return Boolean value for every respective element |
| clip() | Used to clip value below and above to passed Least and Max value |
| clip_lower() | Used to clip values below a passed least value |
| clip_upper() | Used to clip values above a passed maximum value |
| astype() | Method is used to change data type of a series |
| tolist() | Method is used to convert a series to list |
| get() | Method is called on a Series to extract values from a Series. This is alternative syntax to the traditional bracket syntax |
| unique() | Pandas unique() is used to see the unique values in a particular column |
| nunique() | Pandas nunique() is used to get a count of unique values |
| value_counts() | Method to count the number of the times each unique value occurs in a Series |
| factorize() | Method helps to get the numeric representation of an array by identifying distinct values |
| map() | Method to tie together the values from one object to another |
| between() | Pandas between() method is used on series to check which values lie between first and second argument |
| apply() | Method is called and feeded a Python function as an argument to use the function on every Series value. This method is helpful for executing custom operations that are not included in pandas or numpy |

**Note:** you can study in detail by using - https://www.geeksforgeeks.org/python-pandas-series/

## 2.6 Exploratory data analysis

Exploratory Data Analysis (EDA) is an approach that is used to analyse the data and discover trends, patterns, or check assumptions in data with the help of statistical summaries and graphical representations. This allows you to get a better feel of your data and find useful patterns in it.

**Types of EDA:**

Depending on the number of columns we are analysing, we can divide EDA into two types.

A. **Univariate Analysis –** In univariate analysis, we analyse or deal with only one variable at a time. The analysis of univariate data is thus the simplest form of analysis since the information deals with only one quantity that changes. It does not deal with causes or

relationships and the main purpose of the analysis is to describe the data and find patterns that exist within it.

B. **Bi-Variate analysis –** This type of data involves two different variables. The analysis of this type of data deals with causes and relationships and the analysis is done to find out the relationship between the two variables.

C. **Multivariate Analysis** – When the data involves three or more variables, it is categorized under multivariate.

Depending on the type of analysis we can also subcategorize EDA into two parts.

- **Non-graphical Analysis** – In non-graphical analysis, we analyse data using statistical tools like mean median or mode or skewness
- **Graphical Analysis** – In graphical analysis, we use visualizations charts to visualize trends and patterns in the data.

**Steps Involved in Exploratory Data Analysis:**

**Step 1: Import Python Libraries**

Import all libraries which are required for our analysis, such as Data Loading, Statistical analysis, Visualizations, Data Transformations, Merge and Joins, etc.

- Pandas and Numpy have been used for Data Manipulation and numerical Calculations
- **Matplotlib and Seaborn** have been used for Data visualizations.

**Step 2: Reading Dataset**

The Pandas library offers a wide range of possibilities for loading data into the pandas DataFrame from files like JSON, .csv, .xlsx, .sql, .pickle, .html, .txt, images etc.

**Analysing the Data**

- The main goal of data understanding is to gain general insights about the data we can use **shape ( ) method** to get the rows and columns in the dataset.
- We can use **head ( )** method that will display only top 5 rows from the top position, while **tail ( )** method can be used for getting rows from bottom.
- **info()** helps to understand the data type and information about data, including the number of records in each column, data having null or not null, Data type, the memory usage of the dataset.
- **Check for Duplication:** we can use **nunique ()** for checking the duplicacy. Duplicated data can be handled or removed based on further analysis.
- **Missing Values Calculation: isnull**() is widely been in all pre-processing steps to identify null values in the data. **data.isnull().sum()** is used to get the number of missing records in each column

**Step 3: Data Reduction**

Some columns or variables can be dropped if they do not add value to our analysis. We can use **drop ( )** method.

**Step 4: Data Cleaning/Wrangling**

Some names of the variables are not relevant and not easy to understand. Some data may have data entry errors, and some variables may need data type conversion**.**

**Step 5: Statistics Summary**

The information gives a quick and simple description of the data. We can include Count, Mean, Standard Deviation, median, mode, minimum value, maximum value, range, standard deviation, etc. describe() function gives all statistics summary of data

**Step 6: Data Transformation**

```
Import pandas as pd
Import numpy as np
# read dataset using pandas
df = pd.read_csv('C:/Users/Lenovo/OneDrive/Desktop/detail.csv')
print(df)          # print whole data set
#print(df.head(5))   # print only top rows using head()
 # get the shape of the data
#df.shape
df.describe(include = 'all')
df.info()   # to get the information about the dataset
df.nunique()
#Handling Missing Values
df.isnull().sum()
#fill in the missing values
df["Gender"].fillna("No Gender", inplace = True)
 df.isnull().sum()
import seaborn as sns
import matplotlib.pyplot as plt
 sns.histplot(x='Salary', data=df, )
plt.show()
sns.boxplot( x="Salary", y='Team', data=df, )
plt.show()
# Scatter Boxplot For Data Visualization
sns.scatterplot( x="Salary", y='Team', data=df,  hue='Gender', size='Bonus %')
# Placing Legend outside the Figure
plt.legend(bbox_to_anchor=(1, 1), loc=2)
 plt.show()
sns.pairplot(df, hue='Gender', height=2)
```

## 3.1 Reading files

The Pandas library offers a wide range of possibilities for loading data into the pandas DataFrame from files like JSON, .csv, .xlsx, .sql, .pickle, .html, .txt, images etc.

We can use read_csv( ) method for reading the datset.

**Syntax:** df = pd.read_csv ('C:/Users/Lenovo/OneDrive/Desktop/detail.csv')

### 3.2 Data cleaning in python: Handling missing values and filling NA

Data cleaning refers to the process of removing unwanted variables and values from your dataset. It may be removing missing values, outliers, and unnecessary rows/ columns. Re-indexing and reformatting our data. We can use **data.isnull ( ).sum ( )** to get the number of missing records in each column.

### 3.3 Data preparation and pre-processing

Data pre-processing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.

**Why do we need Data Pre-processing?**

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data pre-processing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.

There are following steps involved in the data pre-processing:

- **Getting the dataset**
- **Importing libraries**
- **Importing datasets**
- **Finding Missing Data**
- **Encoding Categorical Data**
- **Splitting dataset into training and test set**
- **Feature scaling**

### 3.4 Data validation techniques in python

Data validation is a process in which we checks the data to make sure it meets some rules or restrictions. There various data validation checks we need to do as follow:

- Is of correct data type, for example a number and not a string
- Does not have invalid values, such as provided zip code has a letter
- Is not out of range, such as age given as negative number or there is division by zero
- Meets constraints, for example, given date can only be in future or message does not exceed maximum length
- Is consistent with other data or constraint, for example student test score and student letter grade are consistent

- Is valid, such as given filename is for an existing file
- Is complete, such as making sure all required form fields have data

**Types of Validation in Python**

There are three types of validation in python, they are:

**Type Check**: This validation technique in python is used to check the given input data type. For example, int, float, etc.

**Length Check**: This validation technique in python is used to check the given input string's length.

**Range Check:** This validation technique in python is used to check if a given number falls in between the two numbers.

**Data Validation Tools**

**1. Cerberus**

While working on data, data validation is a crucial task which ensures that the data is cleaned, corrected and is useful. Cerberus is an open source data validation and transformation tool for Python. The library provides powerful and lightweight data validation functionality which can be easily extensible along with custom validation. The Cerberus 1.x versions can be used with Python 2 while version 2.0 and later rely on Python 3 features.

**2. Colander**

Colander is a Python Library for validating and deserializing data which is obtained via XML, JSON, an HTML form post or any other equally simple data serialisation. It can be said as a good basis for form generation systems, data description systems, and configuration systems. The library has been tested on Python version 2.7 and above and can be used to define a data schema, serialise an arbitrary Python structure to a data structure composed of strings, mappings, and lists and deserialise a data structure composed of strings, mappings, and lists into an arbitrary Python structure after validating the data structure against a data schema.

**3. Schema**

Schema is a library for validating Python data structures such as those obtained from config-files, forms, external services or command-line parsing, converted from JSON/YAML (or something else) to Python data-types. If the data is valid, Schema. Validate will return the validated data and if the data is invalid, Schema will raise SchemaError exception.

**Steps to Data Validation**

**Step 1: Determine data sample**

Determine the data to sample. If you have a large volume of data, you will probably want to validate a sample of your data rather than the entire set. You'll need to decide what volume of data to sample, and what error rate is acceptable to ensure the success of your project.

**Step 2: Validate the database**

Before you move your data, you need to ensure that all the required data is present in your existing database. Determine the number of records and unique IDs, and compare the source and target data fields.

**Step 3: Validate the data format**

Determine the overall health of the data and the changes that will be required of the source data to match the schema in the target. Then search for incongruent or incomplete counts, duplicate data, incorrect formats, and null field values.

**Challenges in Data Validation**

There may be the various key challenge in data validation:

- Validating the database can be challenging because data may be distributed in multiple databases across your organization. The data may be siloed, or it may be outdated.
- Validating the data format can be an extremely time-consuming process, especially if you have large databases and you intend to perform the validation manually. However, sampling the data for validation can help to reduce the time needed.

## 3.5 Data feature engineering: removing columns and rows from raw data

- **Python Pandas Drop Function**

  Pandas **drop** is a function in Python pandas used to drop the rows or columns of the dataset. This function is often used in data cleaning.

  **Note:** axis = 0 is referred as rows and axis = 1 is referred as columns.

**Data Visualization on using Matplotlib and Seaborn libraries**

1. **Data Visualization and its importance**

Data visualization is the representation of data through use of common graphics, such as charts, plots, infographics, and even animations. Data visualizations help us understand the underlying within a dataset or explore the relationships among variables. The importance of Data visualization is – analyzing complex data, identifying patterns, and extracting valuable insights. There are various data visualization tools as Microsoft Power BI, Tableau, Qlik Sense, Klipfolio. Looker, Zoho Analytics, Domo etc.

2. **Libraries used for data visualization**

Python provides various libraries that come with different features for visualizing data. All these libraries come with different features and can support various types of graphs.

- Matplotlib
- Seaborn
- Bokeh
- Plotly

3. **Matplotlib Libraray**

Matplotlib is an easy-to-use, low-level data visualization library that is built on NumPy arrays. It consists of various plots like scatter plot, line plot, histogram, etc. Matplotlib provides a lot of flexibility. Matplotlib is a powerful data visualization tool that allows you to view large amounts of data in comprehensible forms. It was launched by the John Hunter in 2002. We can use the below command to install it: **pip install Matplotlib**

**Advantages of Matplotlib library:**

1. **Matplotlib provides a simple way to access large amounts of data**

   Developers can create accurate plots based on huge amounts of data. When analyzing these data plots, good developers will make it easier to see patterns and trends in the data sets.

2. **It is flexible and supports various forms of data representation**

   Matplotlib supports data representation in bar charts, graphs, scattered plots, and other forms of visualization.

3. **It is easy to navigate**

   The Matplotlib platform isn't too complex. Hence, both beginners and advanced developers can apply their programming skills to the platform, producing professional results. Matplotlib also has subplots that further facilitate the creation and comparison of data sets.

4. **It ensures accessibility by providing high-quality images**

The main goal of Matplotlib is to provide a way to access and display data, its plots and images must be of high quality. To meet this requirement, Matplotlib provides high-quality images in various formats, such as PDF, PGF, and PNG.

5. **It is a powerful tool with numerous applications**

   Matplotlib data-visualization qualities can be used in various forms, such as Python scripts, shells, web application servers, and Jupyter notebooks.

6. **It is open-source**

   An open-source platform requires no paid license. Because Matplotlib is free to use, you save the extra cost you usually incur when producing data visualizations.

7. **It can run on different platforms**

   Matplotlib is platform-independent. This means it can run smoothly no matter what platform you use. Whether your developers use Windows, Mac OS, or Linux, you can expect high-quality results.

8. **It makes data analysis easier**

   Due to its numerous features, plot styles, and high-quality results, Matplotlib makes data analysis easier and more efficient. It also helps save the time and resources you would have spent analyzing large datasets.

**Disadvantages of Matplotlib library**

Matplotlib is excellent for drumming up charts and graphics but it may not be the best fit for time series data. There are few disadvantages as listed below:

1. library is not the best for dealing with time series data since one needs to import all these helper classes for the year, month, week, day formatters, shared a user on a forum.
2. Library is very low-level, which means that one needs to write more code to get the visualization as opposed to Tableau where you can achieve that in a few clicks.

**Seaborn Library**

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

Seaborn is an amazing visualization library for statistical graphics plotting in Python. It provides beautiful default styles and color palettes to make statistical plots more attractive. It is built on top matplotlib library and is also closely integrated with the data structures from pandas.

**Different categories of plot in Seaborn**

Plots are basically used for visualizing the relationship between variables. Those variables can be either completely numerical or a category like a group, class, or division.

**Seaborn divides the plot into the below categories –**

- **Relational plots:** This plot is used to understand the relation between two variables.
- **Categorical plots:** This plot deals with categorical variables and how they can be visualized.
- **Distribution plots:** This plot is used for examining univariate and bivariate distributions
- **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.
- **Matrix plots:** A matrix plot is an array of scatterplots.

**Importing Libraries**

```
import seaborn as sb
```

**Importing Datasets:** load_dataset()

```
df = sb.load_dataset('tips')
```

**Seaborn splits the Matplotlib parameters into two groups−**

- Plot styles

  **Seaborn Figure Styles**

  The interface for manipulating the styles is set_style(). Using this function you can set the theme of the plot. As per the latest updated version, below are the five themes available.

  - Darkgrid
  - Whitegrid
  - Dark
  - White
  - Ticks

```
import numpy as np
from matplotlib import pyplot as plt
def sinplot(flip=1):
  x = np.linspace(0, 14, 100)
  for i in range(1, 5):
    plt.plot(x, np.sin(x + i * .5) * (7 - i) * flip)
import seaborn as sb
sb.set_style("whitegrid")
sinplot()
plt.show()
```

**Removing Axes Spines**

In the white and ticks themes, we can remove the top and right axis spines using the despine() function.

- **Plot scale**

  We have four preset templates for contexts, based on relative size, the contexts are named as follows

  - Paper
  - Notebook
  - Talk
  - Poster

**Seaborn - Color Palette**

Seaborn provides a function called **color_palette(),** which can be used to give colors to plots and adding more aesthetic value to it.

seaborn.color_palette(palette = None, n_colors = None, desat = None)

**Parameter**

The following table lists down the parameters for building color palette −

| Sr.No. | Palatte & Description |
|---|---|
| 1 | **n_colors** Number of colors in the palette. If None, the default will depend on how palette is specified. By default the value of **n_colors** is 6 colors. |
| 2 | **Desat** Proportion to desaturate each color. |

**I. Return**

Return refers to the list of RGB tuples. Following are the readily available Seaborn palettes −

- Deep
- Muted
- Bright
- Pastel
- Dark
- Colorblind

**Seaborn Histogram:**

Histograms represent the data distribution by forming bins along the range of the data and then drawing bars to show the number of observations that fall in each bin.

**Seaborn - Kernel Density Estimates**

Kernel Density Estimation (KDE) is a way to estimate the probability density function of a continuous random variable. It is used for non-parametric analysis.

Setting the **hist** flag to False in **distplot** will yield the kernel density estimation plot.

**Fitting Parametric Distribution**

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

**Scatter Plot**

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

**Example**

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

**II. seaborn.pairplot(data,…)**

**Parameters**

Following table lists down the parameters for Axes −

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

**Key plots using seaborn**

**Histplot:** Seaborn Histplot is used to visualize the univariate set of distributions (single variable). It plots a histogram, with some other variations like kdeplot and rugplot. The Histplot function takes several arguments but the important ones are

- **data**: This is the array, series, or dataframe that you want to visualize. It is a required parameter.
- **x**: This specifies the column in the data to use for the histogram. If your data is a dataframe, you can specify the column by name.
- **y**: This specifies the column in the data to use for the histogram when you want to create a bivariate histogram. By default, it is set to **None**, meaning that a univariate histogram will be plotted.
- **bins**: This specifies the number of bins to use when dividing the data into intervals for plotting. By default, it is set to “auto”, which uses an algorithm to determine the optimal number of bins.
- **kde**: This parameter controls whether to display a kernel density estimate (KDE) of the data in addition to the histogram. By default, it is set to **False**, meaning that a KDE will not be plotted.

```
import numpy as np
import seaborn as sns
 sns.set(style="green")
 # Generate a random univariate dataset
rs = np.random.RandomState(15)
```

```
d = rs.normal(size=80)
 # Plot a simple histogram and kde
sns.histplot(d, kde=True, color="b")
```

**2.Distplot**: Seaborn **distplot** is used to visualize the univariate set of distributions (Single features) and plot the histogram with some other variations like kdeplot and rugplot.

The function takes several parameters, but the most important ones are:

- **a**: This is the array, series, or list of data that you want to visualize. It is a required parameter.
- **bins**: This specifies the number of bins to use when dividing the data into intervals for plotting. By default, it is set to “auto”, which uses an algorithm to determine the optimal number of bins.
- **kde**: This parameter controls whether to display a kernel density estimate (KDE) of the data in addition to the histogram. By default, it is set to **True**, meaning that a KDE will be plotted.
- **hist**: This parameter controls whether to display the histogram of the data. By default, it is set to **True**, meaning that a histogram will be plotted.

**Lineplot:** The line plot is one of the most basic plots in the seaborn library. This plot is mainly used to visualize the data in the form of some time series, i.e. in a continuous manner.

```
import seaborn as sns
 sns.set(style="dark")
fmri = sns.load_dataset("fmri")
 # Plot the responses for different events and regions
sns.lineplot(x="timepoint",
             y="signal",
             hue="region",
             style="event",
             data=fmri)
```

**Functional magnetic resonance imaging (fMRI)** is an imaging scan that shows activity in specific areas of the brain. In medical settings, fMRI mainly helps plan brain surgeries and similar procedures.2

## Seaborn - Statistical Estimation

### Bar Plot

The **barplot()** shows the relation between a categorical variable and a continuous variable. The data is represented in rectangular bars where the length the bar represents the proportion of the data in that category.

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('titanic')
sb.barplot(x = "sex", y = "survived", hue = "class", data = df)
plt.show()
```

### Point Plots

Point plots serve same as bar plots but in a different style. Rather than the full bar, the value of the estimate is represented by the point at a certain height on the other axis.

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('titanic')
sb.pointplot(x = "sex", y = "survived", hue = "class", data = df)
plt.show()
```

## Seaborn - Plotting Wide Form Data

It is always preferable to use 'long-from' or 'tidy' datasets. But at times when we are left with no option rather than to use a 'wide-form' dataset, same functions can also be applied to "wide-form" data in a variety of formats, including Pandas Data Frames or two-dimensional NumPy arrays. These objects should be passed directly to the data parameter the x and y variables must be specified as strings

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
sb.boxplot(data = df, orient = "h")
plt.show()
```

**Seaborn - Multi Panel Categorical Plots**

Categorical data can we visualized using two plots, you can either use the functions **pointplot()**, or the higher-level function **factorplot()**.

**Factorplot**

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

**What is Facet Grid?**

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

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

**Seaborn - Linear Relationships**

There are two main functions in Seaborn to visualize a linear relationship determined through regression. These functions are regplot() and lmplot().

| regplot | lmplot |
|---|---|
| accepts the x and y variables in a variety of formats including simple numpy arrays, pandas Series objects, or as references to variables in a pandas DataFrame | has data as a required parameter and the x and y variables must be specified as strings. This data format is called "long-form" data |

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

**Seaborn - Pair Grid**

PairGrid allows us to draw a grid of subplots using the same plot type to visualize data.

```
import pandas as pd
import seaborn as sb
from matplotlib import pyplot as plt
df = sb.load_dataset('iris')
g = sb.PairGrid(df)
g.map(plt.scatter);
plt.show()
```

**Different categories of plot in Seaborn:** Plots are basically used for visualizing the relationship between variables. Those variables can be either completely numerical or a category like a group, class, or division. Seaborn divides the plot into the below categories –

A. **Relational plots:** This plot is used to understand the relation between two variables.
B. **Categorical plots:** This plot deals with categorical variables and how they can be visualized. There are some key categories plots are like Barplot, Countplot , Boxplot , Violinplot, Stripplot, Swarmplot , Factorplot.
C. **Distribution plots:** This plot is used for examining univariate and bivariate distributions. It has various categories like : Displot, Joinplot ,
D. **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. It has also two categories like Simple linear plot, displaying multiple plots.
E. **Matrix plots:** A matrix plot is an array of scatterplots.
F. **Multi-plot grids:** It is a useful approach to draw multiple instances of the same plot on different subsets of the dataset.

**Advantages of Seaborn Libraries:**

1. By using the Seaborn library, we can easily represent our data on a plot.
2. This library is used to visualize our data. No need to take care of the internal details; we just have to pass our data set or data inside the **relplot ()** function, and it will calculate and place the value accordingly.
3. We can switch to any other representation of data using the 'kind' property inside it.
4. It creates an interactive and informative plot to representation our data; also, this is easy for the user to understand and visualize the records on the application.
5. It uses static aggregation for plot generation in python.

**Disadvantages of Seaborn Libraries:**

1. **Complexity for Beginners:** Seaborn's extensive range of plot types and customization options can be overwhelming for beginners. Users who are new to data visualization or programming might find it challenging to grasp and utilize all of its features effectively.
2. **Limited Customization**: While Seaborn makes it easy to create aesthetically pleasing plots with minimal code, it might be limiting for users who require highly customized and unique visualizations. Advanced customizations might require working directly with the underlying Matplotlib functions.
3. **Learning Curve:** While Seaborn simplifies many aspects of data visualization, there's still a learning curve, especially if users want to take full advantage of its capabilities. Understanding the relationships between different plot functions, using various parameters, and navigating the documentation can take some time.
4. **Performance:** Seaborn is built on top of Matplotlib, which can sometimes result in performance bottlenecks when dealing with very large datasets. While Seaborn tries to optimize its performance, it might not be as efficient as some other libraries for handling big data visualization tasks.
5. **Integration with Other Libraries:** While Seaborn works well with Pandas DataFrames, it might not seamlessly integrate with other data manipulation libraries or tools, which could be a drawback for users who rely on different data processing workflows.
6. **Few Built-in Statistical Tests:** Seaborn provides some basic statistical functionalities like correlation matrix visualizations and simple linear regression plots. However, it might lack the breadth and depth of statistical tests and visualizations that specialized statistical packages like R's ggplot2 offer.

**Difference between Matplotlib and Seaborn Library**

| Matplotlib | Seaborn |
|---|---|
| Matplotlib creates simple graphs, including bar graphs, histograms, pie charts, scatter plots, lines, and other visual representations of data. | There are numerous patterns and graphs for data visualization in Seaborn. It employs engaging themes, and it helps in the integration of all data into a single plot. Additionally, it offers data distribution. |
| It utilizes syntax that is relatively complicated and extensive. Example: Matplotlib.pyplot.bar (x-axis, y-axis) is the syntax for a bar graph. | It has relatively simple syntax, making it simpler to learn and comprehend. Example: seaborn.barplot(x axis, y-axis) syntax for a bar graph. |
| We can open and work with many figures at once. You can close the current figure using the syntax:matplotlib.pyplot.close(). Close all the figures using this syntax: matplotlib.pyplot.close("all") | Seaborn sets the time for the creation of each figure. However, it may lead to (OOM) memory issues. |
| Matplotlib is a Python graphics package for data visualization and integrates nicely with Numpy and Pandas. Similar capabilities and syntax are available in Pyplot as in MATLAB, and users of MATLAB can readily understand it. | Seaborn is more comfortable with Pandas data frames. It utilizes simple sets of techniques to produce lovely images in Python. |
| Matplotlib is highly customized and robust. | With the help of its default themes, Seaborn prevents overlapping plots. |
| Matplotlib plots various graphs using Pandas and Numpy. | Seaborn is the extended version of Matplotlib, which uses Matplotlib, Numpy, and Pandas to plot graphs. |
| Seaborn is licensed under the GNU GPL. | Matplotlib is licensed under the Python Software Foundation License. |

### 4. Scatter plot

A scatter plot is a graph in which the values of two variables are plotted along two axes, the pattern of the resulting points revealing any correlation present.

Scatterplot can be used with several semantic. They can plot two-dimensional graphics that can be enhanced by mapping up to three additional variables while using the semantics of hue, size, and style parameters.

**Syntax:** seaborn.scatterplot(x=None, y=None, hue=None, style=None, size=None, data=None, palette=None, hue_order=None, hue_norm=None, sizes=None, size_order=None,

size_norm=None, markers=True, style_order=None, x_bins=None, y_bins=None, units=None, estimator=None, ci=95, n_boot=1000, alpha='auto', x_jitter=None, y_jitter=None, legend='brief', ax=None, **kwargs)

**Parameters:**

**X, y:** Input data variables that should be numeric.

**Data:** Data frame where each column is a variable and each row is an observation.

**Size:** Grouping variable that will produce points with different sizes.

**Style:** Grouping variable that will produce points with different markers.

**Palette:** Grouping variable that will produce points with different markers.

**Markers:** Object determining how to draw the markers for different levels.

**Alpha:** Proportional opacity of the points.

**Returns:** This method returns the Axes object with the plot drawn onto it.

```
# importing required modules
import pandas as pd
import numpy as np
import seaborn
# loading the dataset
dataset = seaborn.load_dataset('tips')
# heading of the dataset
#print(dataset.head(20))
seaborn.set(style='whitegrid')
tip = seaborn.load_dataset('tips')
seaborn.scatterplot(x='day', y='tip', data=tip)
#Adding the marker attributes
seaborn.scatterplot(x='day', y='tip', data= tip, marker = '+')
#Adding the hue attributes.
seaborn.scatterplot(x='day', y='tip', data=tip, hue='time')
seaborn.scatterplot(x='day', y='tip', data=tip, hue='day')
#Adding the style attributes.
seaborn.scatterplot(x='day', y='tip', data=tip, hue="time", style="time")
#Adding the palette attributes.
seaborn.scatterplot(x='day', y='tip', data=tip, hue='time', palette='pastel')
#Adding size attributes.
```

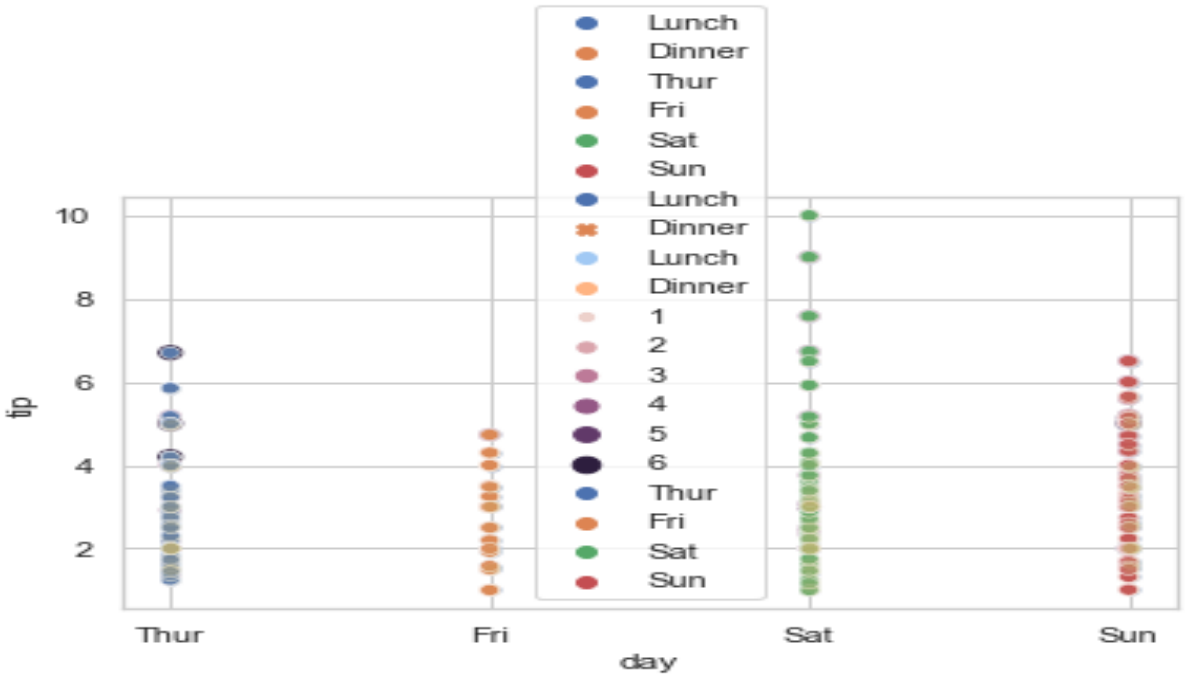
seaborn.scatterplot(x='day', y='tip', data=tip ,hue='size', size = "size")

#Adding legend attributes.

seaborn.scatterplot(x='day', y='tip', data=tip, hue='day',sizes=(30, 200), legend='brief')

#Adding alpha attributes.

seaborn.scatterplot(x='day', y='tip', data=tip, alpha = 0.1)

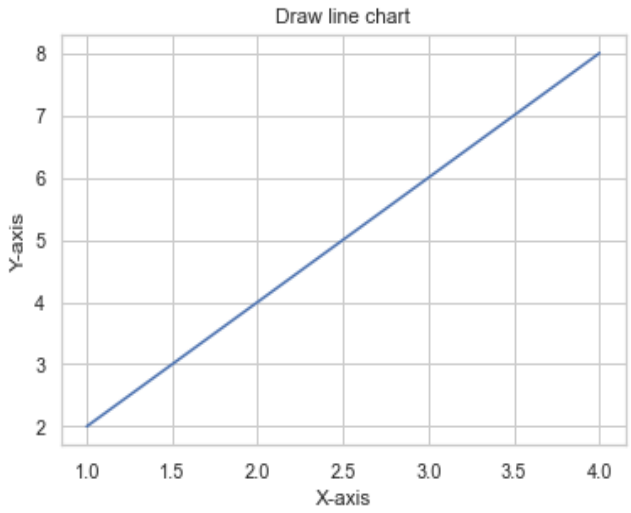



### 5. Line Plot

Line charts are used to represent the relation between two data X and Y on a different axis. We can draw line chart as given below:

- **Single line graph**

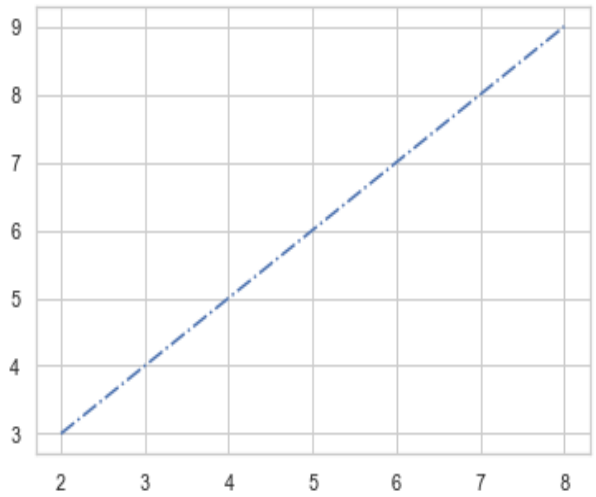
```
Import matplotlib.pyplot as plt
Import numpy as np
# Define X and Y variable data
x = np.array ([1, 2, 3, 4])
y = x*2
plt.plot(x, y)
plt.xlabel("X-axis")  # add X-axis label
plt.ylabel("Y-axis")  # add Y-axis label
plt.title("Draw Line chart")  # add title
plt.show()
```




- **Multiple charts in a line graph**

  We can display more than one chart in the same container by using pyplot.figure() function.

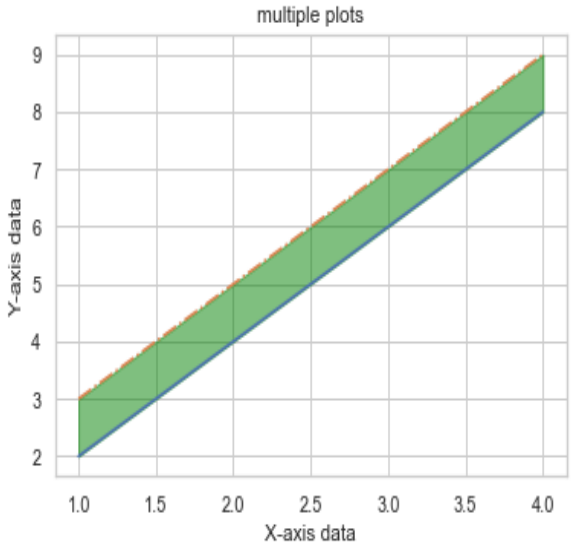
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([1, 2, 3, 4])
y = x*2
plt.plot(x, y)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("My Graph")
plt.show()  # show first chart
# The figure() function helps in creating a
# new figure that can hold a new chart in it.
plt.figure()
x1 = [2, 4, 6, 8]
y1 = [3, 5, 7, 9]
plt.plot(x1, y1, '-.')
# Show another chart with '-' dotted line
plt.show()
```




❖ **Fill the area between two plots**

Using the pyplot.fill_between () function we can fill in the region between two line plots in the same graph.

```
Import matplotlib.pyplot as plt
Import numpy as np
 x = np.array ([1, 2, 3, 4])
y = x*2
plt.plot(x, y)
x1 = [2, 4, 6, 8]
y1 = [3, 5, 7, 9]
plt.plot(x, y1, '-.')
plt.xlabel("X-axis data")
plt.ylabel("Y-axis data")
plt.title('multiple plots')
plt.fill_between(x, y, y1, color='green', alpha=0.5)
plt.show()
```



### 6. Bar Plot

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically.

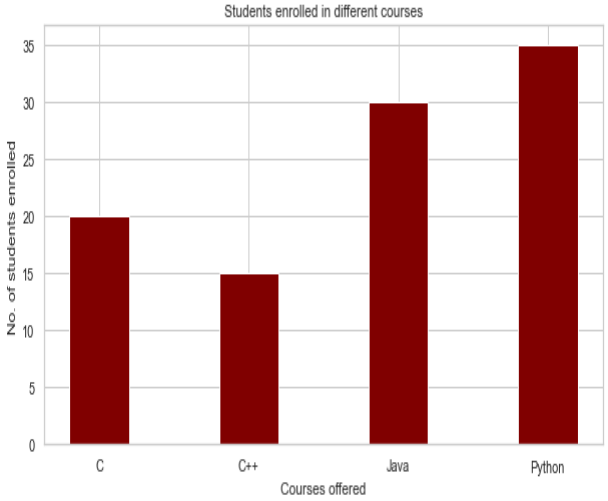
A bar chart describes the comparisons between the discrete categories. One of the axis of the plot represents the specific categories being compared, while the other axis represents the measured values corresponding to those categories.

❖ **Creating a bar plot:**

The matplotlib API in Python provides the **bar ( )** function which can be used in MATLAB style use or as an object-oriented API. The syntax of the bar() function to be used with the axes is as follows:-

**Syntax:** plt.bar(x, height, width, bottom, align)

```
import numpy as np
import matplotlib.pyplot as plt
# creating the dataset
data = {'C':20, 'C++':15, 'Java':30, 'Python':35}
courses = list(data.keys())
values = list(data.values())
    fig = plt.figure(figsize = (10, 5))
     # creating the bar plot
    plt.bar(courses, values, color ='maroon',width = 0.4)
     plt.xlabel("Courses offered")
    plt.ylabel("No. of students enrolled")
    plt.title("Students enrolled in different courses")
    plt.show()
```




❖ **Multiple bar plots**

Multiple bar plots are used when comparison among the data set is to be done when one variable is changing. We can easily convert it as a stacked area bar chart, where each subgroup is displayed by one on top of the others. It can be plotted by varying the thickness and position of the bars**.**

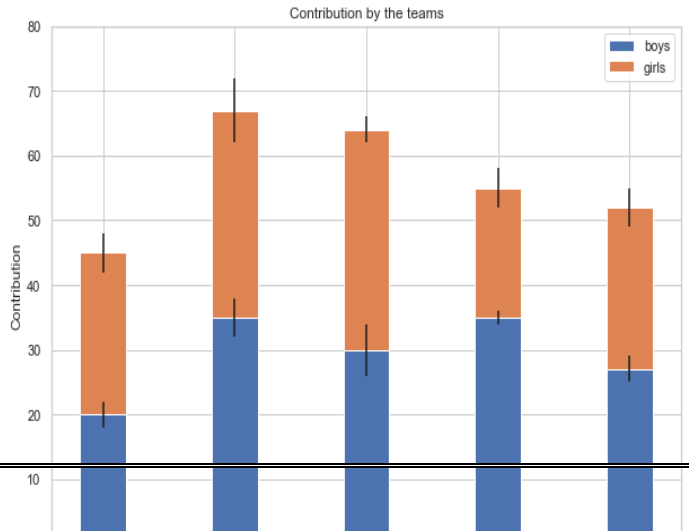
```
import numpy as np
import matplotlib.pyplot as plt
    # set width of bar
    barWidth = 0.25
```

```
fig = plt.subplots(figsize =(12, 8))
# set height of bar
IT = [12, 30, 1, 8, 22]
ECE = [28, 6, 16, 5, 10]
CSE = [29, 3, 24, 25, 17]
# Set position of bar on X axis
br1 = np.arange(len(IT))
br2 = [x + barWidth for x in br1]
br3 = [x + barWidth for x in br2]
# Make the plot
plt.bar(br1, IT, color ='r', width = barWidth,
edgecolor ='grey', label ='IT')
plt.bar(br2, ECE, color ='g', width = barWidth,
edgecolor ='grey', label ='ECE')
plt.bar(br3, CSE, color ='b', width = barWidth,
edgecolor ='grey', label ='CSE')
# Adding Xticks
plt.xlabel('Branch', fontweight ='bold', fontsize = 15)
plt.ylabel('Students passed', fontweight ='bold', fontsize = 15)
plt.xticks([r + barWidth for r in range(len(IT))],['2015', '2016', '2017', '2018', '2019'])
plt.legend()
plt.show()
```

- **Stacked bar plot**

Stacked bar plots represent different groups on top of one another. The height of the bar depends on the resulting height of the combination of the results of the groups. It goes from the bottom to the value instead of going from zero to value.

```
import numpy as np
import matplotlib.pyplot as plt
N = 5
boys = (20, 35, 30, 35, 27)
girls = (25, 32, 34, 20, 25)
boyStd = (2, 3, 4, 1, 2)
girlStd = (3, 5, 2, 3, 3)
```

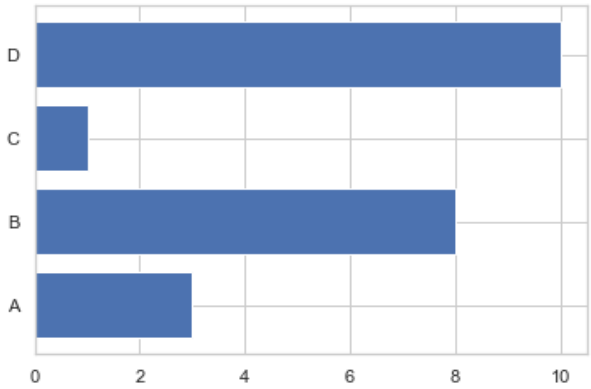


```
ind = np.arange(N)
width = 0.35
fig = plt.subplots(figsize =(10, 7))
p1 = plt.bar(ind, boys, width, yerr = boyStd)
p2 = plt.bar(ind, girls, width,  bottom = boys, yerr = girlStd)
plt.ylabel('Contribution')
plt.title('Contribution by the teams')
plt.xticks(ind, ('T1', 'T2', 'T3', 'T4', 'T5'))
plt.yticks(np.arange(0, 81, 10))
plt.legend((p1[0], p2[0]), ('boys', 'girls'))
plt.show()
```

- **Horizontal Bars**

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

```
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()
```




### 7. Histogram

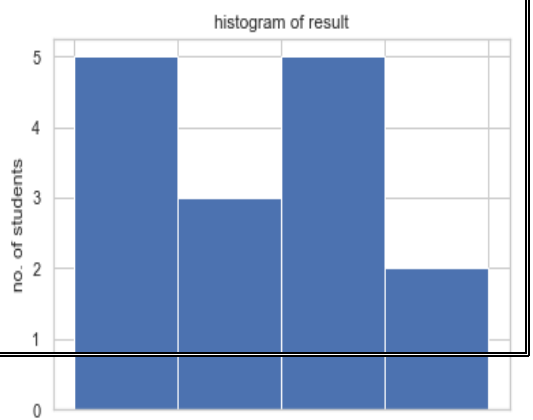
A histogram is an accurate representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable. It is a kind of bar graph.

**To construct a histogram, we follow these steps –**

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

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

```
from matplotlib import pyplot as plt
import numpy as np
fig,ax = plt.subplots(1,1)
a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27])
ax.hist(a, bins = [0,25,50,75,100])
ax.set_title("histogram of result")
```



```
ax.set_xticks([0,25,50,75,100])
ax.set_xlabel('marks')
ax.set_ylabel('no. of students')
plt.show()
```

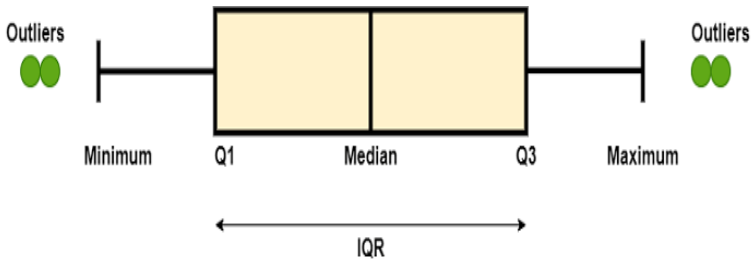
### 8. Box plot

A Box Plot is also known as Whisker plot is created to display the summary of the set of data values having properties like minimum, first quartile, median, third quartile and maximum.

A Box plot is a way to visualize the distribution of the data by using a box and some vertical lines. It is known as the whisker plot.

The data can be distributed between five key ranges, which are as follows:

- Minimum: Q1-1.5*IQR
- 1st quartile (Q1): 25th percentile
- Median:50th percentile
- 3rd quartile(Q3):75th percentile
- Maximum: Q3+1.5*IQR

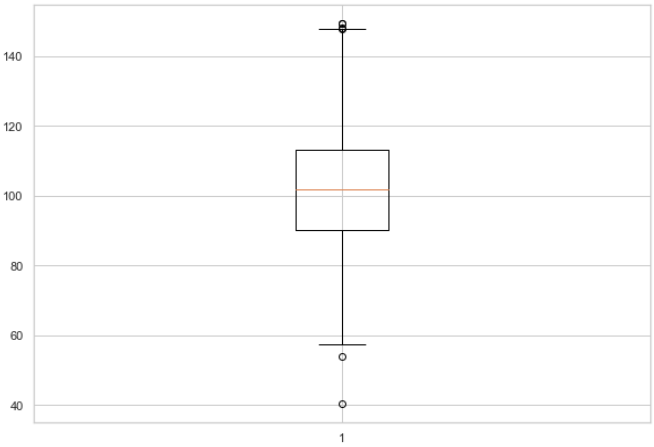



**Creating Box Plot**

The **matplotlib.pyplot** module of **Matplotlib** library provides boxplot () function with the help of which we can create box plots.

**Syntax:** matplotlib.pyplot.boxplot(data, notch=None, vert=None, patch_artist=None, widths=None)

```
import matplotlib.pyplot as plt
import numpy as np
 # Creating dataset
np.random.seed(10)
data = np.random.normal(100, 20, 200)
fig = plt.figure(figsize =(10, 7))
# Creating plot
plt.boxplot(data)
# show plot
plt.show()
```

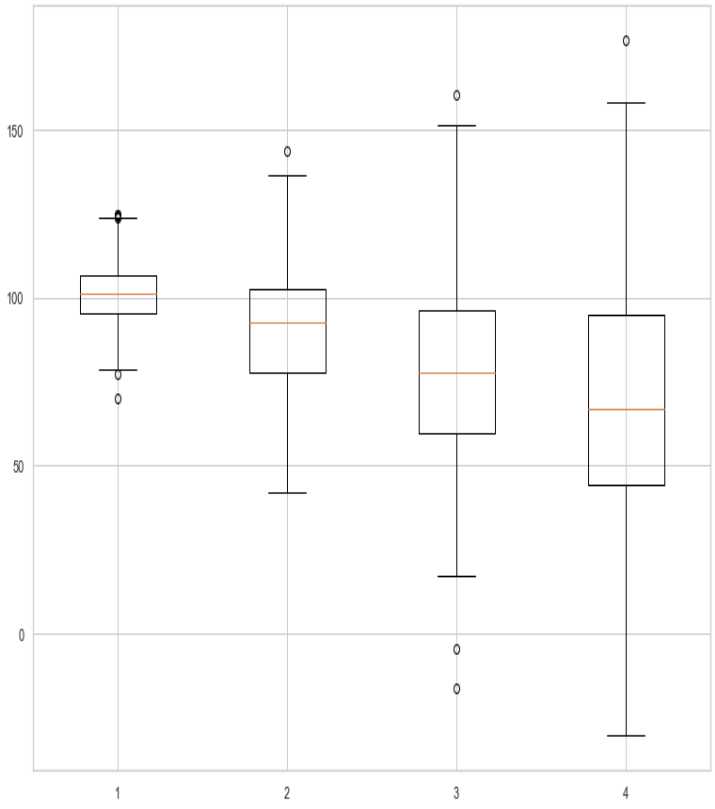



- **Customizing Box Plot**

The matplotlib.pyplot.boxplot() provides endless customization possibilities to the box plot. The notch = True attribute creates the notch format to the box plot, patch_artist = True fills the boxplot with colors, we can set different colors to different boxes.

The vert = 0 attribute creates horizontal box plot. labels takes same dimensions as the number data sets.

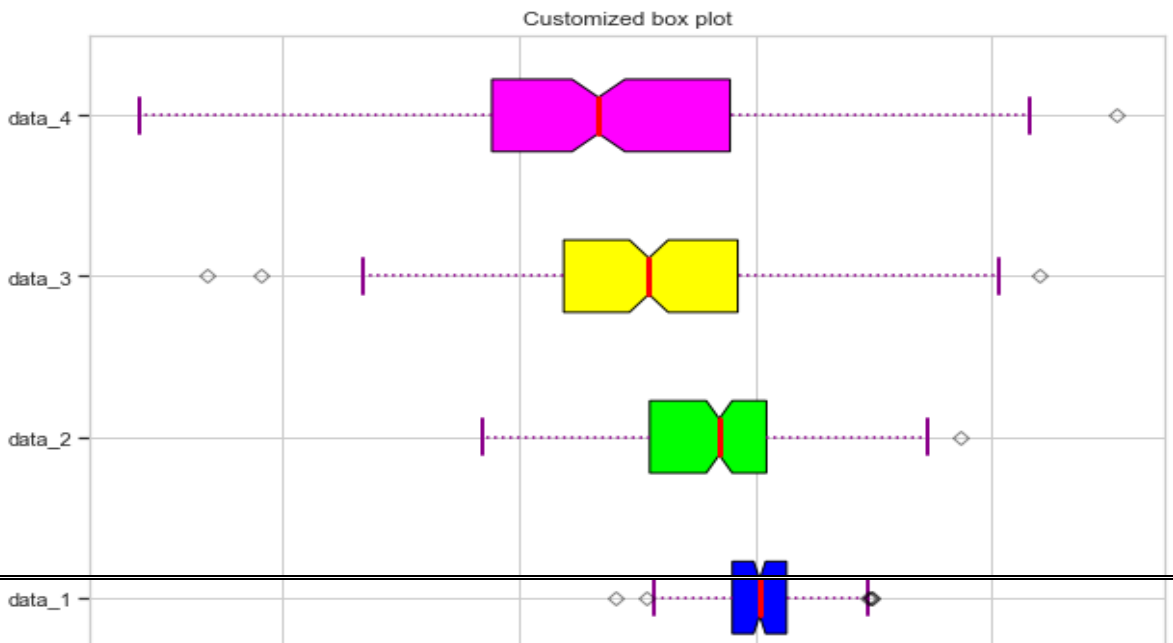
```
import matplotlib.pyplot as plt
import numpy as np
 # creating dataset
np.random.seed(10)
 data_1 = np.random.normal(100, 10, 200)
data_2 = np.random.normal(90, 20, 200)
data_3 = np.random.normal(80, 30, 200)
data_4 = np.random.normal(70, 40, 200)
data = [data_1, data_2, data_3, data_4]
 fig = plt.figure(figsize =(10, 7))
# Creating axes instance
ax = fig.add_axes([0, 0, 1, 1])
 # Creating plot
bp = ax.boxplot(data)
# show plot
plt.show()
```




- **Some of the customizations:**

```
import matplotlib.pyplot as plt
import numpy as np
 # Creating dataset
np.random.seed(10)
data_1 = np.random.normal(100, 10, 200)
data_2 = np.random.normal(90, 20, 200)
data_3 = np.random.normal(80, 30, 200)
data_4 = np.random.normal(70, 40, 200)
data = [data_1, data_2, data_3, data_4]
fig = plt.figure(figsize =(10, 7))
ax = fig.add_subplot(111)
# Creating axes instance
bp = ax.boxplot(data, patch_artist = True,  notch ='True', vert = 0)
colors = ['#0000FF', '#00FF00', '#FFFF00', '#FF00FF']
for patch, color in zip(bp['boxes'], colors): patch.set_facecolor(color)
```

```
 # changing color and linewidth of
# whiskers
for whisker in bp['whiskers']:
    whisker.set(color ='#8B008B',  linewidth = 1.5,  linestyle =":")
    # changing color and linewidth of
    # caps
    for cap in bp['caps']:
     cap.set(color ='#8B008B',  linewidth = 2)
    # changing color and linewidth of
    # medians
    for median in bp['medians']:
       median.set(color ='red', linewidth = 3)
    # changing style of fliers
    for flier in bp['fliers']:
      flier.set(marker ='D', color ='#e7298a', alpha = 0.5)
    # x-axis labels
    ax.set_yticklabels(['data_1', 'data_2',  'data_3', 'data_4'])
    # Adding title
    plt.title("Customized box plot")
    # Removing top axes and right axes
    # ticks
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    # show plot
    plt.show()
```



### 9. Pair plot

The **Seaborn.pairplot ( )** method is used to plot pairwise relationships in a dataset. Each numeric variable in the data will be spread over the y-axes across a single row and the x-axes across a single column by default, according to the axes grid created by this function.

**Syntax:** Following is the syntax of the seaborn.pairplot () method –

seaborn.pairplot(data, *, hue=None, hue_order=None, palette=None, vars=None, x_vars=None, y_vars=None, kind='scatter', diag_kind='auto', markers=None, height=2.5, aspect=1, corner=False, dropna=False, plot_kws=None, diag_kws=None, grid_kws=None, size=None)

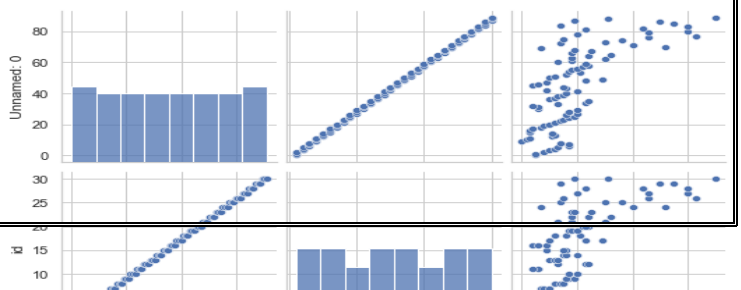
**Parameters**

| S.No | Name and Description |
|---|---|
| 1 | **Data:** Takes dataframe where each column is a variable and each row is an observation. |
| 2 | **Hue:** variables that specify portions of the data that will be displayed on certain grid facets. To regulate the order of levels for this variable, refer to the var order parameters. |
| 3 | **Kind:** Takes values from{'scatter', 'kde', 'hist', 'reg'}and the kind of plot to make is determined. |
| 4 | **Diag_kind: Takes** values from {'auto', 'hist', 'kde', None} and the kind of plot for the diagonal subplots is determined if used. If 'auto', choose based on whether or not hue is used. |
| 5 | **Height:** Takes a scalar value and determines the height of the facet. |
| 6 | **Aspect:** Takes scalar value and Aspect ratio of each facet, so that aspect * height gives the width of each facet in inches. |
| 7 | **Corner:** Takes Boolean value and If True, don't add axes to the upper (off-diagonal) triangle of the grid, making this a "corner" plot. |
| 8 | **hue_order:** Takes lists as input and Order for the levels of the faceting variables is determined by this order. |

**Creation of pair plot:**

import seaborn as sns

import matplotlib.pyplot as plt



```
exercise =sns.load_dataset("exercise")
exercise.head()
sns.pairplot(exercise)
plt.show()
```

**For Example:**

```
import seaborn as sns
import matplotlib.pyplot as plt
exercise =sns.load_dataset("exercise")
exercise.head()
g = sns.pairplot(exercise, diag_kind="kde")
g.map_lower(sns.kdeplot, levels=4, color=".8")
plt.show()
```