PANDAS

1. What is Pandas?

- Pandas or Python Data Analysis Library is the most frequently used, open-source and popular library in python that is mainly used for in depth data analysis.

- Many people jump onto machine learning without having to understand Pandas thoroughly as it provides the ability to process, munge and classify your data.

- In order to understand ML (Machine Learning), you have to have a good grip on pandas. In simple words, pandas work exactly in python how excel works in Microsoft office.

2. History of Pandas

- Pandas was developed by Wes McKinney in 2008 because of the need for an excellent, robust and super fast data analysis tool for data.
- Pandas is declared an open source library for performing data analysis in Python

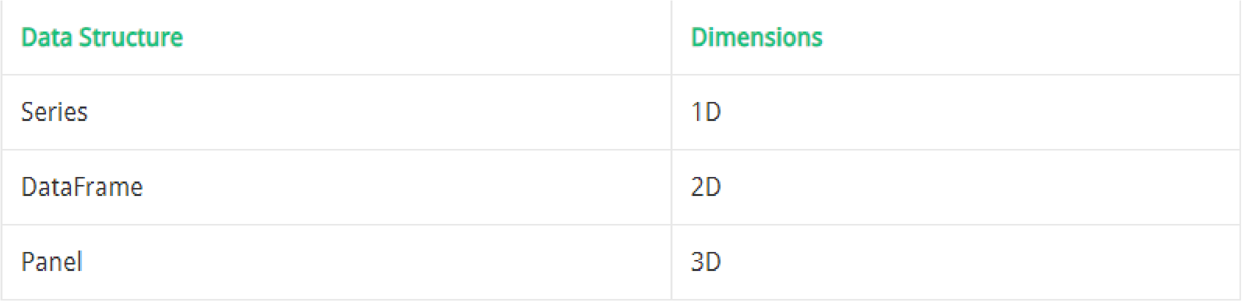
Usage

Pandas can be used in different areas and fields like:

>Statistics
> Space Centers (NASA, ISRO etc)
> Data Centers (for analyzing data)
>Social Media Websites
>FinTech Companies

3. What is Data Structure in Pandas?

- One of the most important things in Pandas is to understand the data structure that it has, once you have mastered it then you can understand How Series, Data frame and Panes are divided.
- Pandas is divided into three data structures when it comes to dimensionality of an array. These data structures are:
  1. Series
  2. DataFrame
  3. Panel



Data Structure	Dimensions
Series	1D
DataFrame	2D
Panel	3D

Series and **Data Frames** are the most widely used data structures based on the usage and problem solving sets in data science.

If we look at these data structures in terms of a spreadsheet then

Series

Would be a single column of an excel sheet, whereas

Data Frame

Will have rows and columns and be a sheet itself.

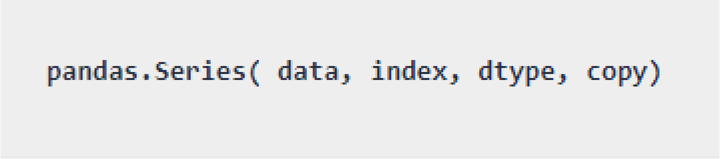
Panel

Will look like a group of sheets which can have multiple Data Frames.

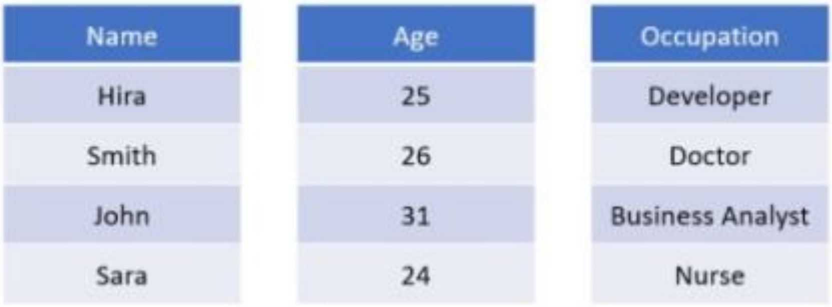
Series Data Structure in Pandas:

- As we have learned, series is a one dimensional data structure that is capable of handling or storing any type of data be it string, number, integer, float, objects, etc.
- Series contains just one axis i.e. of a column, as that axis is labelled as the index of the series.

Syntax of series:-



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



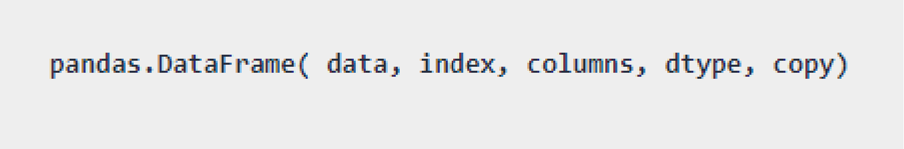
Name	Age	Occupation
Hira	25	Developer
Smith	26	Doctor
John	31	Business Analyst
Sara	24	Nurse

DataFrame Data Structure in Pandas:

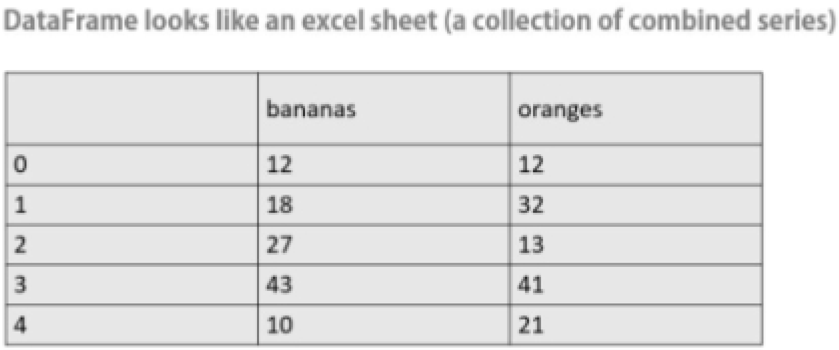
- Dataframe in pandas is one step ahead of series (since it is a one dimensional data structure).
- Dataframe is a 2D data structure having labelled axes as rows and columns.
- In order to create a dataframe, we need to always work around three main aspects:

1. Data (Source to populate our dataframe with)
2. Rows (Horizontal wise)
3. Columns (Vertical wise)

Syntax of Dataframe is:



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



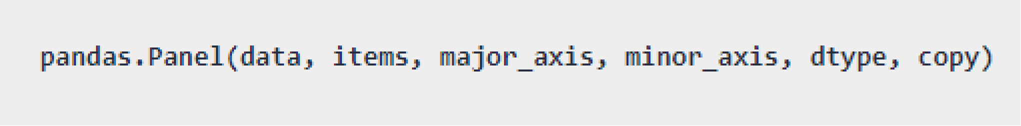
DataFrame looks like an excel sheet (a collection of combined series)

	bananas	oranges
0	12	12
1	18	32
2	27	13
3	43	41
4	10	21

Panel in Pandas

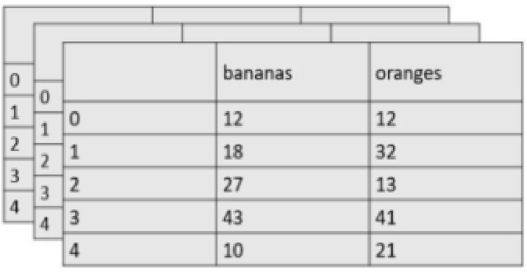
- Panel in pandas is used for working with 3-dimensional data.
- It is not used that much in real world examples. But, let's say that you have sets of dataframes and you want to analyze all of them.
- Then you can use the option of panel in pandas.

Syntax of panel is:



```
pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)
```

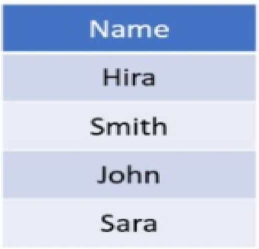
Panel looks like multiple excel sheets



	bananas	oranges
0	12	12
1	18	32
2	27	13
3	43	41
4	10	21

What is a Series in Pandas?

- Pandas series is a one dimensional data structure which can have values of integer, float and string.
- We use series when we want to work with a single dimensional array. It is important to note that series cannot have multiple columns.
- It only holds one column just like in an excel sheet. Series does have an index as an axis label. You can have your own index labels by customizing the index values.

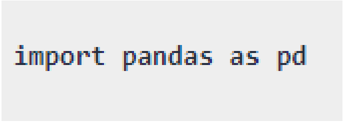


Name
Hira
Smith
John
Sara

This is a series

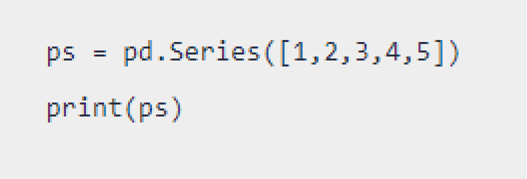
Creating a Series in Pandas:

- Pandas Series can be created in different ways from MySQL table, through excel worksheet (CSV) or from an array, dictionary, list etc.
- Let's look at how to create a series. Let's import Pandas first into the python file or notebook that you are working in:



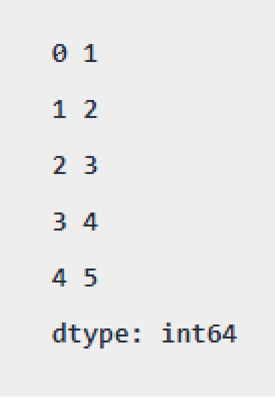
```
import pandas as pd
```

- **After importing pandas as pd, it’s time to use the Series method**
- **Ex:-**



```
ps = pd.Series([1,2,3,4,5])

print(ps)
```

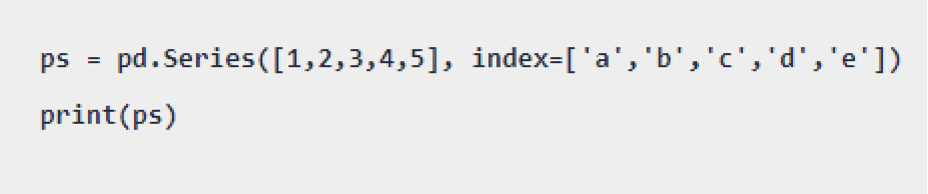
Output:-



```
0 1

1 2

2 3

3 4

4 5

dtype: int64
```

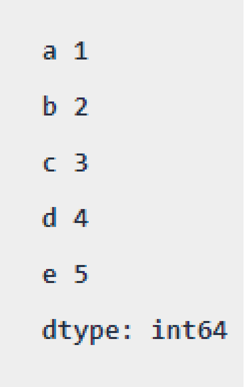
Changing the index of Series in Pandas:

- By default, the index values of your series are numbers ranging from 0 onwards.
- You can change the index of the series by customizing the index values inside a list, in order to achieve that use the index argument to change values.
- **Ex:-**



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

print(ps)
```

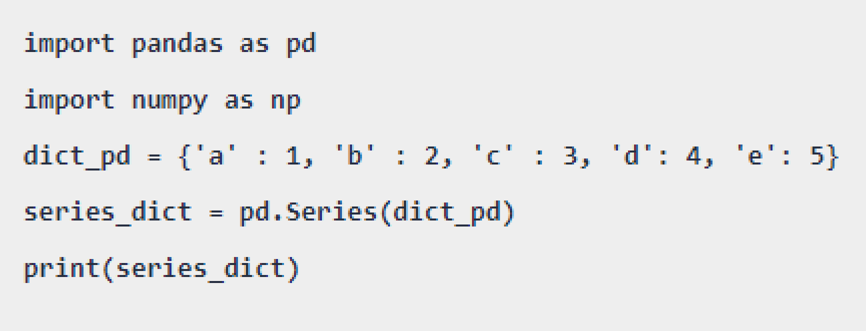
Output:-



```
a 1

b 2

c 3

d 4

e 5

dtype: int64
```

Creating a Series from a Dictionary:

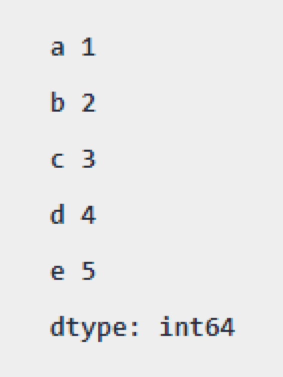
- Let’s learn about creating series from a dictionary, just like creating a conventional Series in Pandas, a dictionary has all the elements predefined to suit a Series.
- If an index is not specified while declaring the Series, then the keys are considered to be index by default.
- If an index is passed then keys are replaced as index labels
- **Ex: -**



```
import pandas as pd

import numpy as np

dict_pd = {'a' : 1, 'b' : 2, 'c' : 3, 'd': 4, 'e': 5}

series_dict = pd.Series(dict_pd)

print(series_dict)
```

.

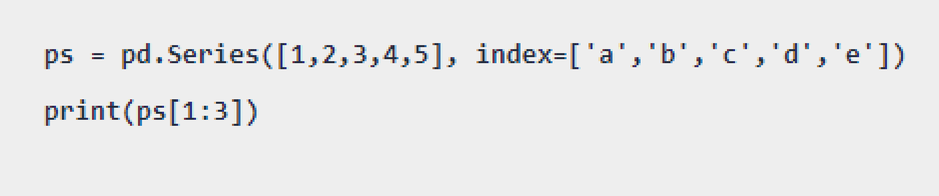
Output:-



```
a 1

b 2

c 3

d 4

e 5

dtype: int64
```

Accessing elements in Series:

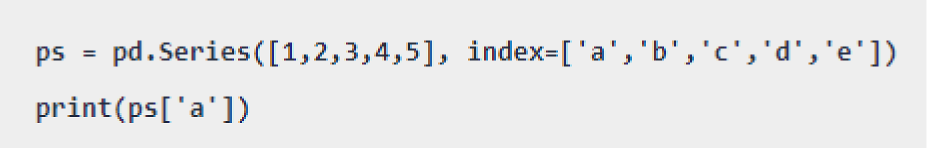
- You can access the elements in Series by using the index position values.
- The index positioning will not change even though we customize the index value. However,
- You can always target the index label itself as well, be it any integer label or a string.
- Ex:-



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

print(ps[1:3])
```

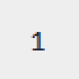
You can target the index label as well, let’s say that we want to pick ‘a’ as our label to test an example:

- **Ex:-**



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

print(ps['a'])
```

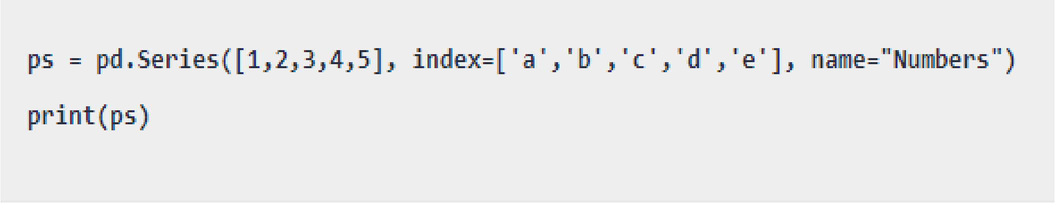
Output:



```
1
```

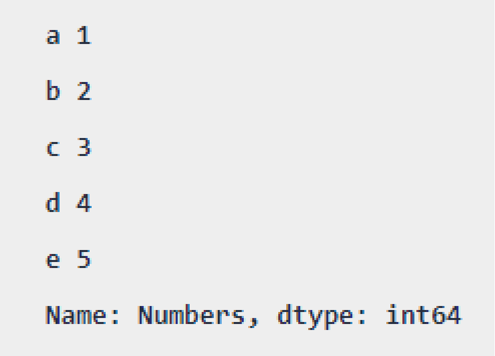
Labeling Series Column

- You can set a name for your Series as well by using the ‘name’ attribute depending on the type of your data.
- **Ex**:-



```
ps = pd.Series([1,2,3,4,5], index=['a','b','c','d','e'], name="Numbers")

print(ps)
```

Output:-



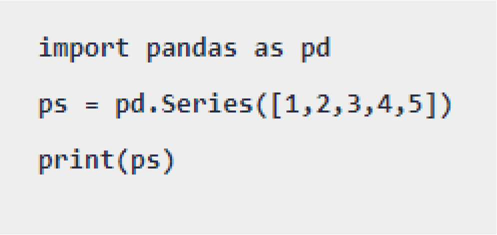
```
a 1

b 2

c 3

d 4

e 5

Name: Numbers, dtype: int64
```

Understanding CRUD in Series:

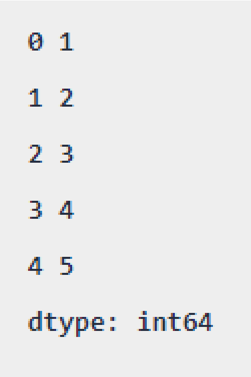
- The pandas series data structure enables us to perform CRUD operations, i.e. **Creating, reading, updating and deleting data**.
- Once you perform these operations or a single operation on a series then a new series is returned.

CRUD: Creating a Series

- Pandas Series can be created in different ways from MySQL table, through excel worksheet (CSV) or from an array, dictionary, list etc.
- Let’s look at how to create a series. Let’s import Pandas first into the python file or notebook that you are working in:
- **Ex:-**



```
import pandas as pd

ps = pd.Series([1,2,3,4,5])

print(ps)
```

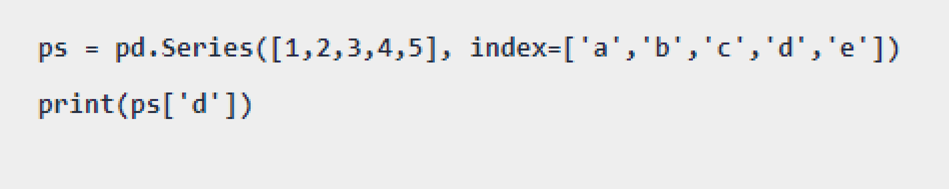


```
0 1
1 2
2 3
3 4
4 5
dtype: int64
```

Output:-

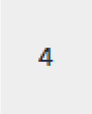
CRUD: Reading in Series

- In order to read and select data from a series, you can use the index attribute by defining the index value or an index position (if no index is defined by you).
- Let's select data through an index value.
- **Ex:-**



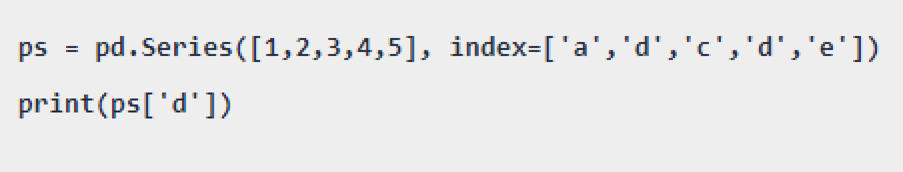
```
ps = pd.Series([1,2,3,4,5], index=['a','b','c','d','e'])
print(ps['d'])
```

Output:-



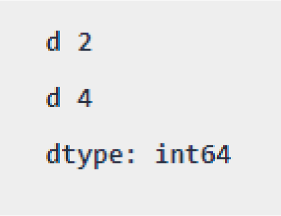
```
4
```

- The above computation returned 4 which is a scalar value, However
- Series object will be returned if index values are repeated.
- **Ex:-**



```
ps = pd.Series([1,2,3,4,5], index=['a','d','c','d','e'])
print(ps['d'])
```

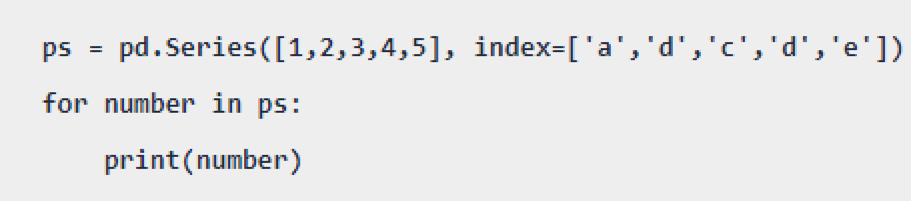
Output:-



```
d 2
d 4
dtype: int64
```

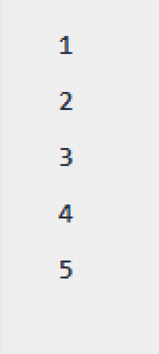
- **Explanation**:-Above, you can see that index values (customized) have'd' repeated twice and it returned the values in series located in those positions.

- You can use a **for loop** as well to read the values in the Series.
- **Ex:-**



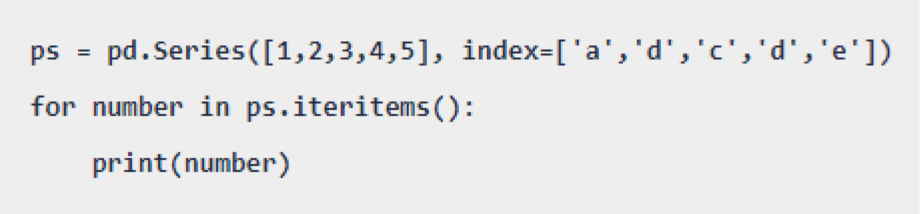
```
ps = pd.Series([1,2,3,4,5], index=['a','d','c','d','e'])
for number in ps:
    print(number)
```

Output:



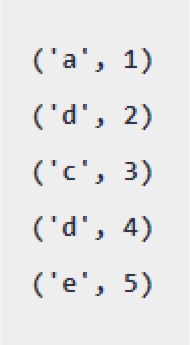
```
1
2
3
4
5
```

- In order to print series values along with their indexes (both default or customized), use the **.iteritems()** which iterate over **(index, value)** tuples method in the for loop
- **Ex:-**



```
ps = pd.Series([1,2,3,4,5], index=['a','d','c','d','e'])
for number in ps.iteritems():
    print(number)
```

Output:-

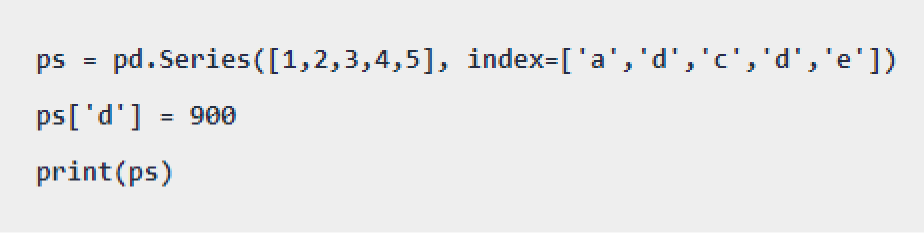


```
('a', 1)
('d', 2)
('c', 3)
('d', 4)
('e', 5)
```

CRUD: Updating in Series:

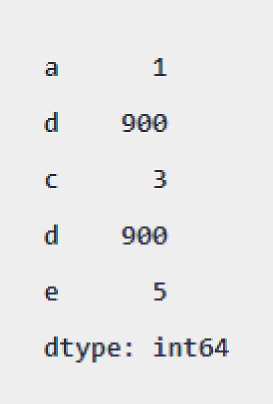
- You can update or replace the values in series as well by selecting the index position or value,

Ex:-



```
ps = pd.Series([1,2,3,4,5], index=['a','d','c','d','e'])

ps['d'] = 900

print(ps)
```

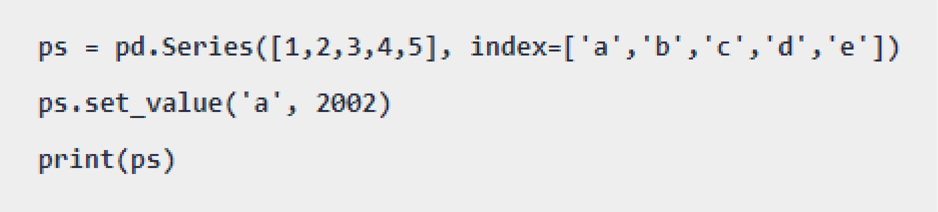
Output:-



```
a      1

d    900

c      3

d    900

e      5

dtype: int64
```

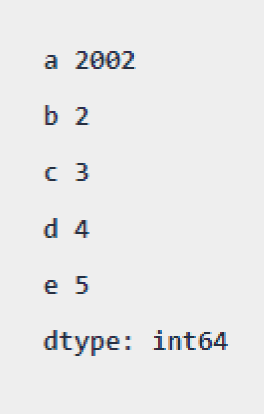
- You can use the **set.value()** method as well by setting the index separated by a comma with a new updated value in that index position.

- **Ex:-**



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

ps.set_value('a', 2002)

print(ps)
```

Output:-

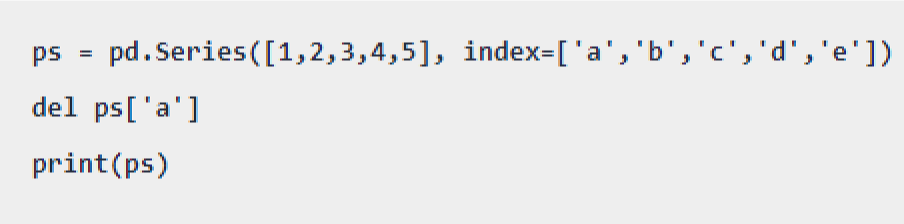


```
a 2002

b 2

c 3

d 4

e 5

dtype: int64
```

Note: the **set_value ()** method is depreciated so it's better to not put it in practice.

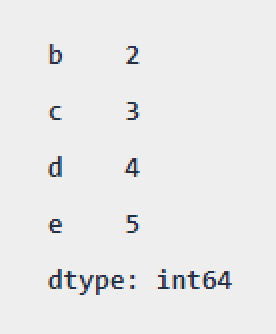
Deleting in Series:

- You can delete an entry in Series by selecting the **del** statement.
- **EX:-**



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

del ps['a']

print(ps)
```

Output:-

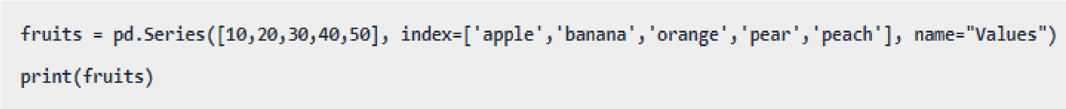


```
b    2

c    3

d    4

e    5

dtype: int64
```

Performing Indexing in Series:-

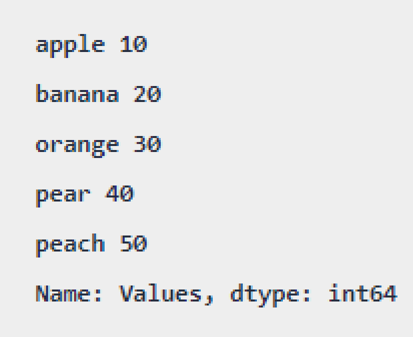
- Discuss in detail about how to perform indexing of Series.
- It is important to understand that our index values don’t have to be whole numbers.
- We can perform indexing on strings as well.

Ex:-



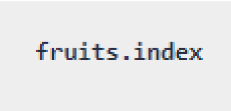
```
fruits = pd.Series([10,20,30,40,50], index=['apple','banana','orange','pear','peach'], name="Values")

print(fruits)
```

Output:-



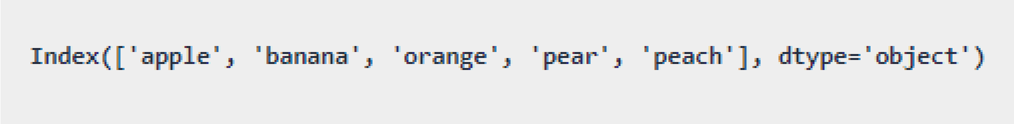
```
apple 10

banana 20

orange 30

pear 40

peach 50

Name: Values, dtype: int64
```

- In order to find the index-only values
- you can use the index function along with the series name and in return you will get all the index values as well as data type of the index
- **Ex:-**



```
fruits.index
```

Output:

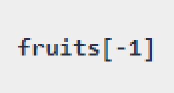


```
Index(['apple', 'banana', 'orange', 'pear', 'peach'], dtype='object')
```

- Above, you can see the data type of the index declared as an '**object**'.
- If the indexes were integers then the datatype would have been **int**.

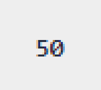
Negative Indexing in Series

- You can also access the element of a Series by adding negative indexing,
  For example:
  To fetch the last element of the Series, you will call **'-1'** as your index position and see what your output is:
- **Ex:-**



```
fruits[-1]
```

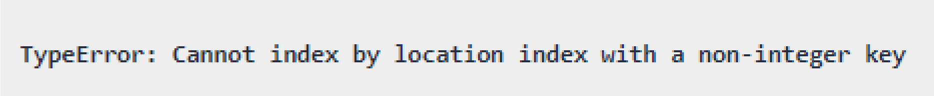
Output:-



```
50
```

iloc and loc indexing in Series

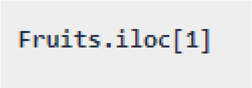
- **iloc and loc** methods are used for indexing labels and index positions respectively. **iloc** method is specifically used for indexing index position and never a label, otherwise an error will pop up as:



```
TypeError: Cannot index by location index with a non-integer key
```

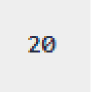
- Whereas, **loc** method is used for indexing only labels, so if you have indexes as strings or strings of even numbers as '12', it's always a good practice to use loc as an index method.

For iloc:



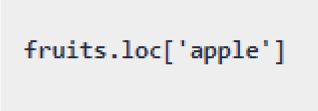
```
Fruits.iloc[1]
```

- **Output:-**



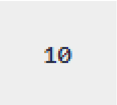
```
20
```

For loc:



```
fruits.loc['apple']
```

- **Output:-**



```
10
```

What is Dataframe in Pandas?

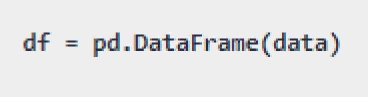
- As we learned that series is a one dimensional data structure, dataframe is the opposite of it as it is two dimensional data structure with labeled axes (rows and columns).
- Whenever we deal with Dataframes, we always keep three things in mind:

→ **Data to populate the dataframe**
→ **Rows**
→ **Columns**

Creating a DataFrame in Pandas

- Pandas Dataframe can be created via arrays, lists, dictionaries, through external storage like **SQL database, CSV** files or excel sheets.
  Hence, there are multiple ways to create a Dataframe.
- We are going to be looking at a few to understand dataframe in a better way.
- We always use a DataFrame notation followed by the parentheses which includes the data.

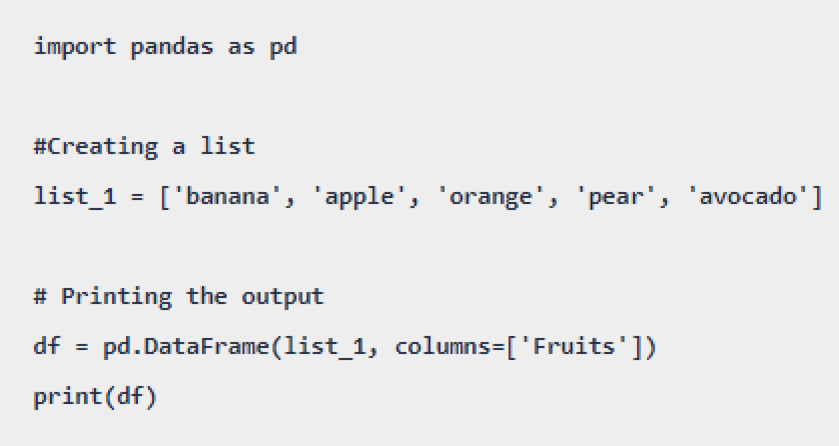
Syntax of using a dataframe is:



```
df = pd.DataFrame(data)
```

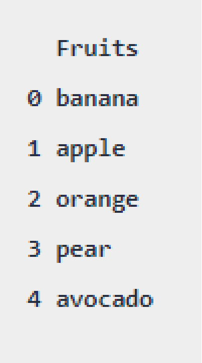
Creating a DataFrame in Pandas via List

- Dataframes can be created through a **list** or a set of lists.
- **Ex:-**



```
import pandas as pd

#Creating a list
list_1 = ['banana', 'apple', 'orange', 'pear', 'avocado']

# Printing the output
df = pd.DataFrame(list_1, columns=['Fruits'])
print(df)
```

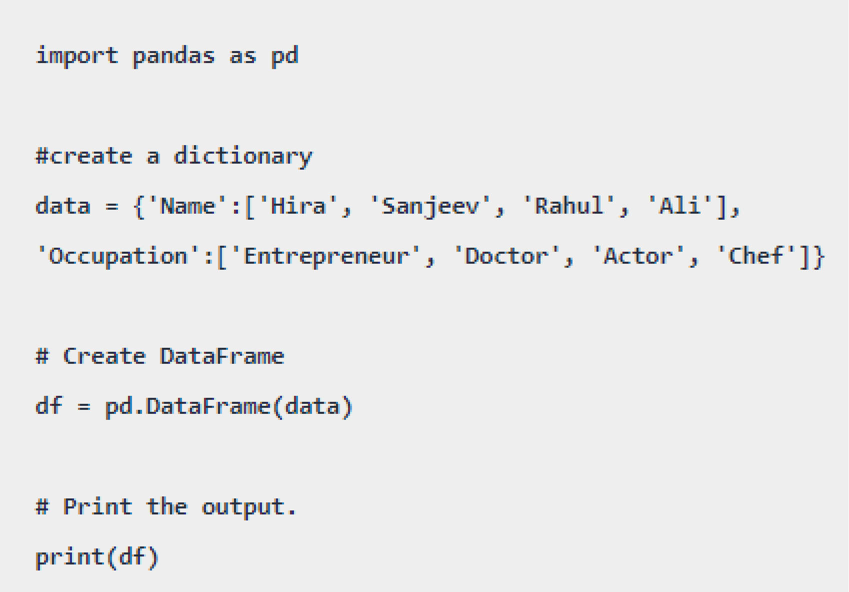
Output:-



```
  Fruits
0 banana
1 apple
2 orange
3 pear
4 avocado
```

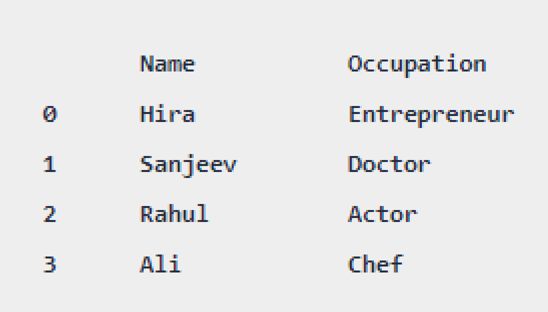
Creating a DataFrame in Pandas via DICTONARY:-

- DataFrame can be created through a dictionary, where keys are going to act as the column names
- **Ex**:-



```
import pandas as pd

#create a dictionary
data = {'Name':['Hira', 'Sanjeev', 'Rahul', 'Ali'],
'Occupation':['Entrepreneur', 'Doctor', 'Actor', 'Chef']}

# Create DataFrame
df = pd.DataFrame(data)

# Print the output.
print(df)
```

OUTPUT:-

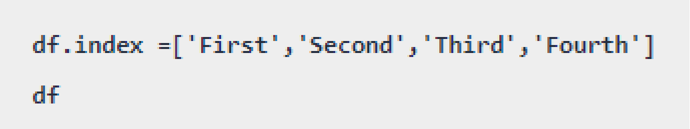


```
        Name        Occupation
0       Hira        Entrepreneur
1       Sanjeev     Doctor
2       Rahul       Actor
3       Ali         Chef
```

➔ **Note**: Indexing and slicing works in the same way in **DataFrame** as it worked in a Series.

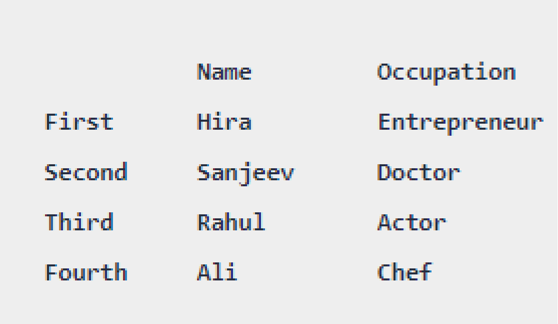
Indexing DataFrame in Pandas

- Like series, indexes are set as integers (starting from 0) by default; however, you can set your own indexes as well by using the index method.
- **Ex:-**



```
df.index =['First','Second','Third','Fourth']
df
```

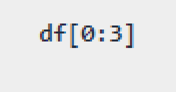
Output:-



```
             Name        Occupation
First        Hira        Entrepreneur
Second       Sanjeev     Doctor
Third        Rahul       Actor
Fourth       Ali         Chef
```

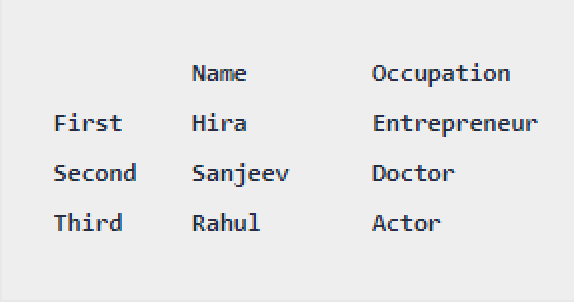
Slicing a DataFrame

- Slicing a dataframe is as simple as slicing a Series or a regular list in python, let’s say that you want to retrieve a few required rows from your dataframe.
- You can **slice** the dataframe by passing the index positions of your rows.
- **Ex:-**



```
df[0:3]
```

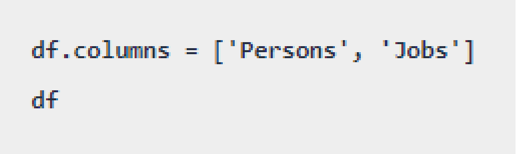
- **Output:-**



```
         Name       Occupation
First    Hira       Entrepreneur
Second   Sanjeev    Doctor
Third    Rahul      Actor
```

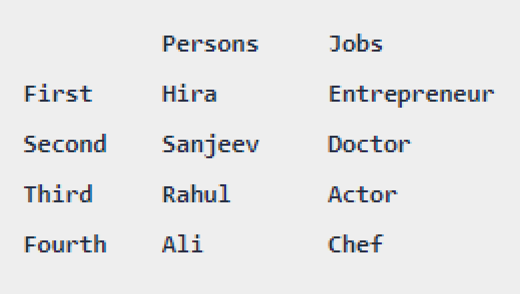
Modifying the Column Value:-

- You can change the name of your column as well,
  For example
  - ➔ In some cases especially when you use dictionary then keys by default become your column names in DataFrame. To change that, you can use the column attribute
- **Ex:-**



```
df.columns = ['Persons', 'Jobs']
df
```

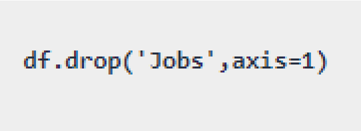
- **Output:-**



```
         Persons    Jobs
First    Hira       Entrepreneur
Second   Sanjeev    Doctor
Third    Rahul      Actor
Fourth   Ali        Chef
```

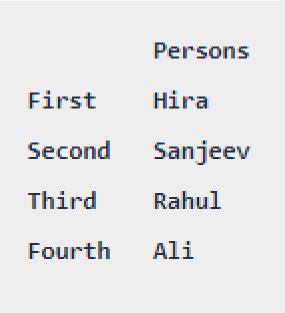
Dropping Rows and Columns

- You can delete rows and columns from your dataframe as well by selecting the name of the row and defining the axis where rows and columns are placed (axis 0 is for rows and axis 1 is for columns).
- Let’s say we want to remove ‘Jobs’ from our columns so we will use the drop method to do
- **Ex:-**



```
df.drop('Jobs',axis=1)
```

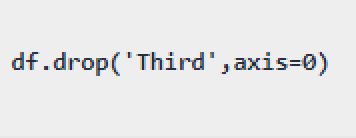
- **Output:-**



```
        Persons
First   Hira
Second  Sanjeev
Third   Rahul
Fourth  Ali
```

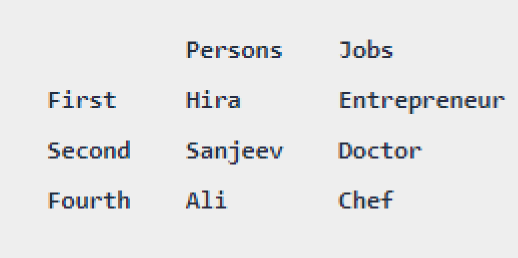
Note:

- **We can use the drop method for rows as well;**
- **Let’s say we want to eliminate the third row, so we will initiate:**
- **Ex:-**



```
df.drop('Third',axis=0)
```

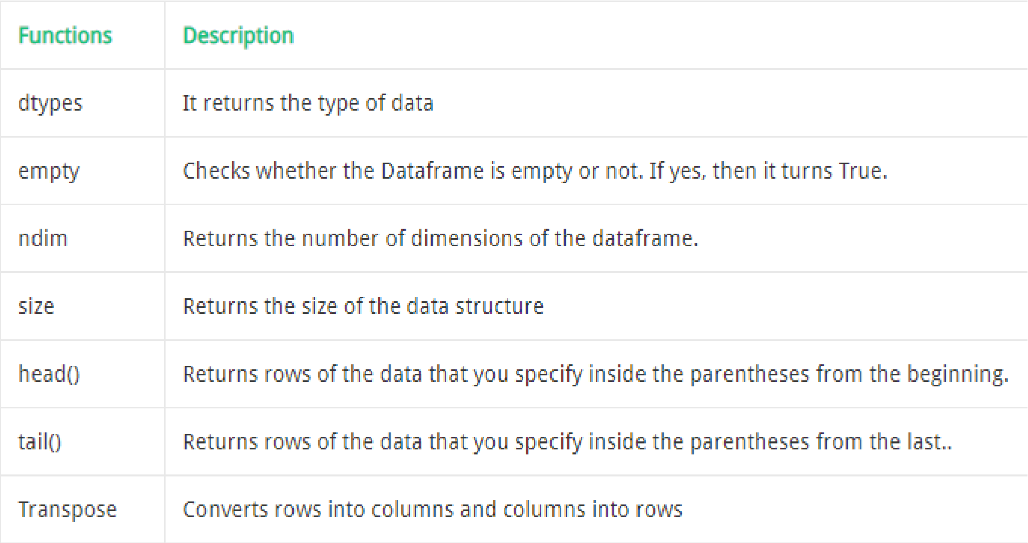
- **Output :-**



```
        Persons   Jobs
First   Hira      Entrepreneur
Second  Sanjeev   Doctor
Fourth  Ali       Chef
```

Understanding Functions in Pandas

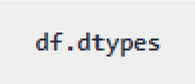
- By now we know how to create different types of data structures in Pandas.
- We have learned about creating a Series and a DataFrame.
- Now it’s time to learn about different functionalities in Pandas to perform different tasks.



Functions	Description
dtypes	It returns the type of data
empty	Checks whether the Dataframe is empty or not. If yes, then it turns True.
ndim	Returns the number of dimensions of the dataframe.
size	Returns the size of the data structure
head()	Returns rows of the data that you specify inside the parentheses from the beginning.
tail()	Returns rows of the data that you specify inside the parentheses from the last..
Transpose	Converts rows into columns and columns into rows

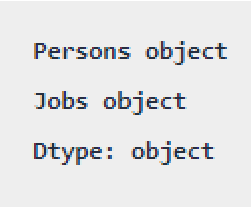
Functions in Pandas: dtypes

- It returns the type of data.
- **Ex:-**



```
df.dtypes
```

- **Output:**

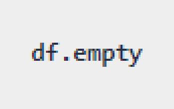


```
Persons object
Jobs object
Dtype: object
```

-

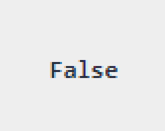
Functions in Pandas: empty

- Checks whether the Dataframe is empty or not. If yes, then it turns true.
- **Ex:-**



```
df.empty
```

- **Output:-**

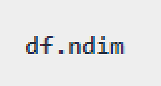


```
False
```

- **Explanation: -** Since our dataframe is not empty hence empty returned False.

Functions in Pandas: ndim

- Returns the number of dimensions of the dataframe.
- **Ex:-**

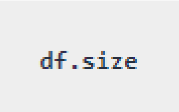


```
df.ndim
```

- **Output:-** 2

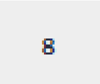
Functions in Pandas: size

- Returns the size of the data structure (number of rows and columns):
- **Ex:-**



```
df.size
```

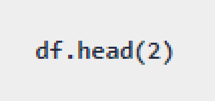
- **Output:-**



```
8
```

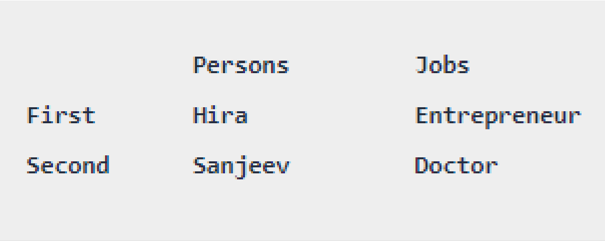
head():

- Returns rows of the data that you specify inside the parentheses from the beginning.
- **Ex:-**



```
df.head(2)
```

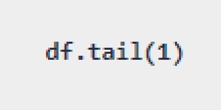
- **Output:-**



```
          Persons    Jobs
First     Hira       Entrepreneur
Second    Sanjeev    Doctor
```

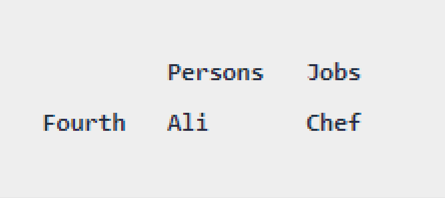
tail()

- Returns rows of the data that you specify inside the parentheses.
- **Ex:-**



```
df.tail(1)
```

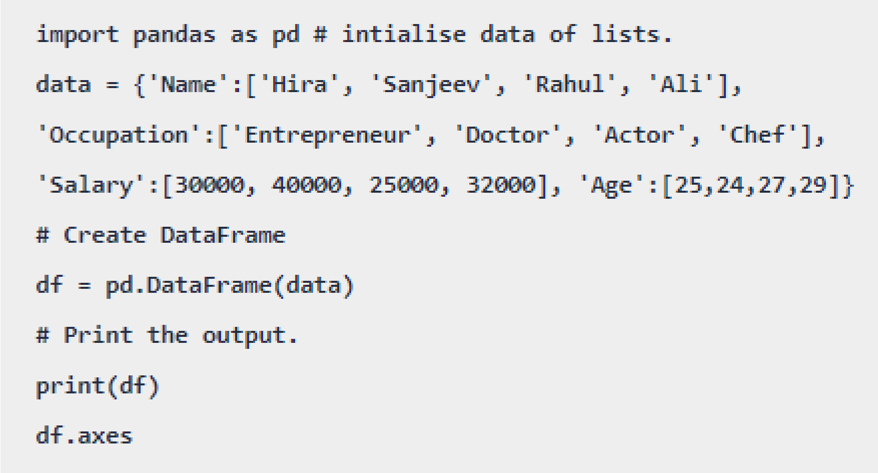
- **Output:-**



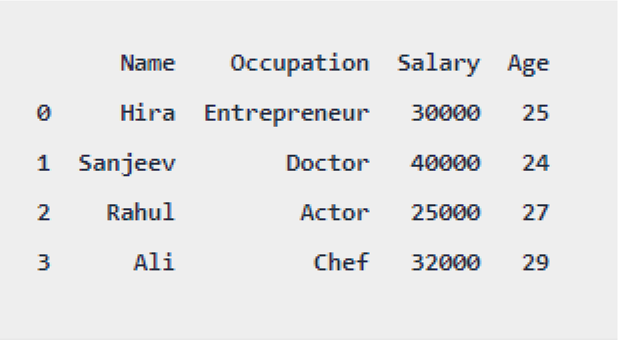
```
         Persons   Jobs
Fourth   Ali       Chef
```

Axes

- Axes function returns the rows axis label and column axis label.
- **Ex:-**

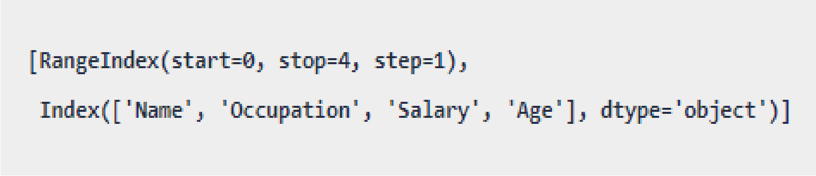


```
import pandas as pd # intialise data of lists.
data = {'Name':['Hira', 'Sanjeev', 'Rahul', 'Ali'],
'Occupation':['Entrepreneur', 'Doctor', 'Actor', 'Chef'],
'Salary':[30000, 40000, 25000, 32000], 'Age':[25,24,27,29]}
# Create DataFrame
df = pd.DataFrame(data)
# Print the output.
print(df)
df.axes
```



```
      Name    Occupation  Salary  Age
0     Hira  Entrepreneur   30000   25
1  Sanjeev        Doctor   40000   24
2    Rahul         Actor   25000   27
3      Ali          Chef   32000   29
```

- **output:-**



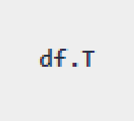
```
[RangeIndex(start=0, stop=4, step=1),
 Index(['Name', 'Occupation', 'Salary', 'Age'], dtype='object')]
```

Explanation:

- Above, you can see that we are able to create axis labels of rows and columns by simply using the axes function.
- It is displaying the range index as well as a separated index from the dictionary keys.

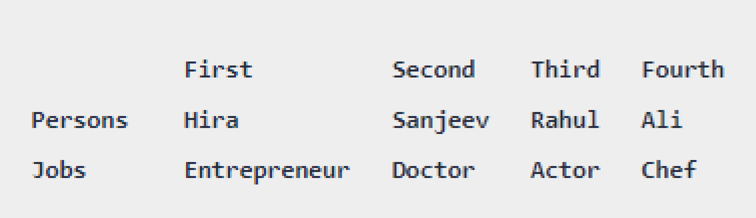
Transpose

- Converts rows into columns and columns into rows.



```
df.T
```

- **Ex:-**



```
         First          Second   Third  Fourth
Persons  Hira           Sanjeev  Rahul  Ali
Jobs     Entrepreneur   Doctor   Actor  Chef
```

- **Output:-**

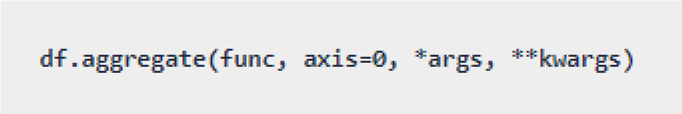
Explanation:

- Rows are converted into columns

Understanding Aggregation in Pandas

- So as we know that pandas is a great package for performing data analysis because of its flexible nature of integration with other libraries.
- The aggregation function is used for one or more rows or columns to aggregate the given type of data.

The syntax of the aggregation function is:

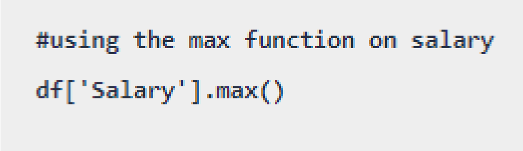


```
df.aggregate(func, axis=0, *args, **kwargs)
```

NOTE:- axis 0 refers to the index values whereas axis 1 refers to the rows.

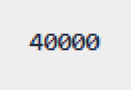
Aggregation in Pandas: Max Function

- EX:-



```
#using the max function on salary
df['Salary'].max()
```

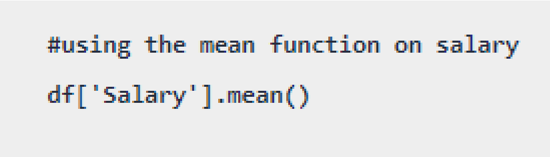
- Output:-



```
40000
```

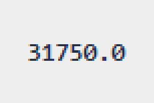
Aggregation in Pandas: Mean Function

- Ex:-



```
#using the mean function on salary
df['Salary'].mean()
```

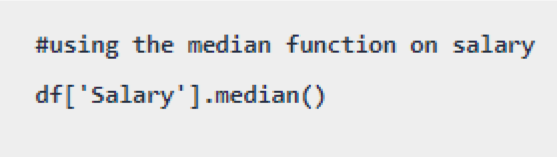
- Output:-



```
31750.0
```

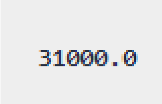
Aggregation in Pandas: Median Function

- Ex:-



```
#using the median function on salary
df['Salary'].median()
```

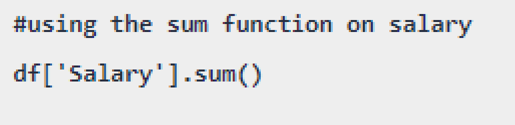
- Output:-



```
31000.0
```

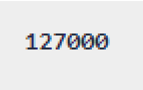
Sum Function:

- Ex:-



```
#using the sum function on salary
df['Salary'].sum()
```

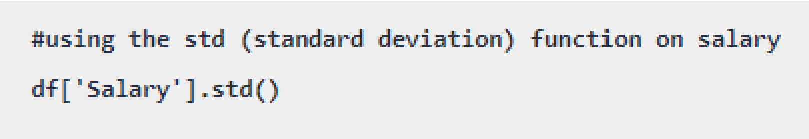
- Output :-



```
127000
```

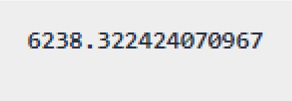
Standard Deviation:

- Ex:-



```
#using the std (standard deviation) function on salary
df['Salary'].std()
```

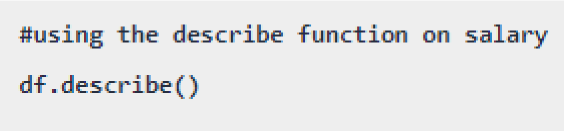
- Output:-



```
6238.322424070967
```

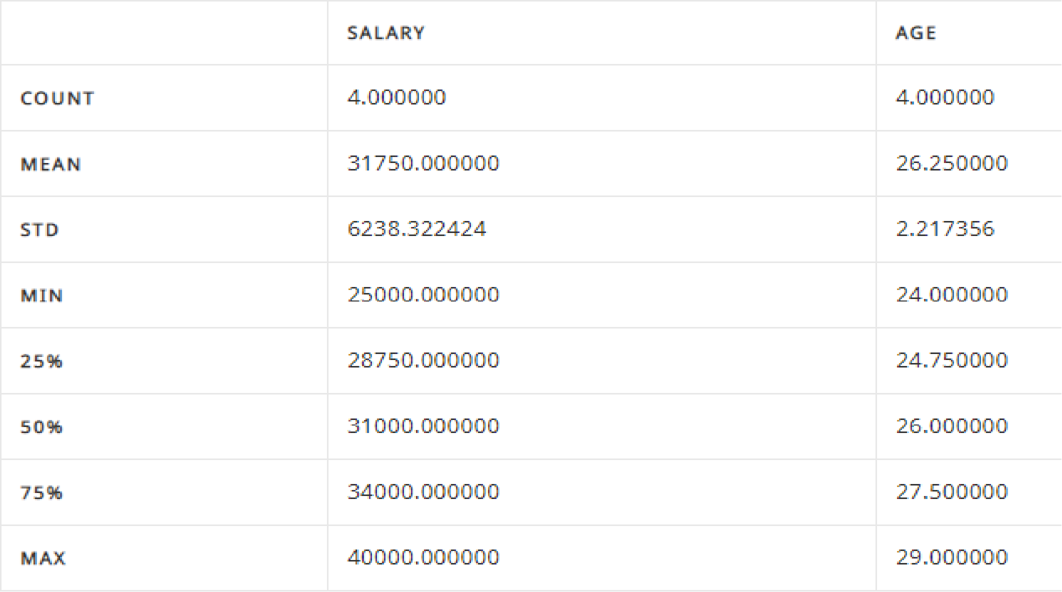
Describe Function:

- **Ex:-**



```
#using the describe function on salary
df.describe()
```

- **Output:-**



	SALARY	AGE
COUNT	4.000000	4.000000
MEAN	31750.000000	26.250000
STD	6238.322424	2.217356
MIN	25000.000000	24.000000
25%	28750.000000	24.750000
50%	31000.000000	26.000000
75%	34000.000000	27.500000
MAX	40000.000000	29.000000

- It covers all the basic aggregation functions **like** sum, max, min, describe, count etc to work around with data.

- Another important aspect of performing or squeezing a dataframe into a selected dataframe is group by where you can classify your own columns and perform aggregation functions through grouping.

What is Groupby in Pandas?

- Pandas is an awesome tool for classifying data into groups through the **groupby() method**. We can distribute the objects in pandas on any of their axis.
- In short, groupby means to analyze a pandas Series by some category.

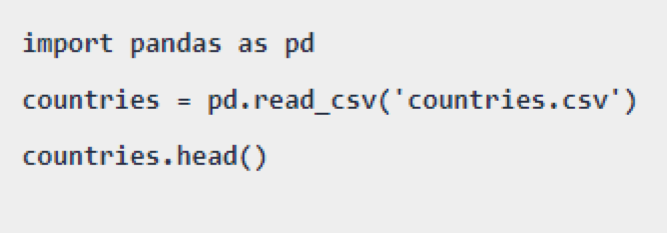
- In short, if you have repeated categories in your dataset, then you can create groups in order to classify your data into sub groups. Remember, it won’t be wise to perform groupby method on unique values.

- Let’s look at the syntax of groupby to understand it in more depth:

  DataFrame.groupby(self, by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, observed=False, **kwargs).

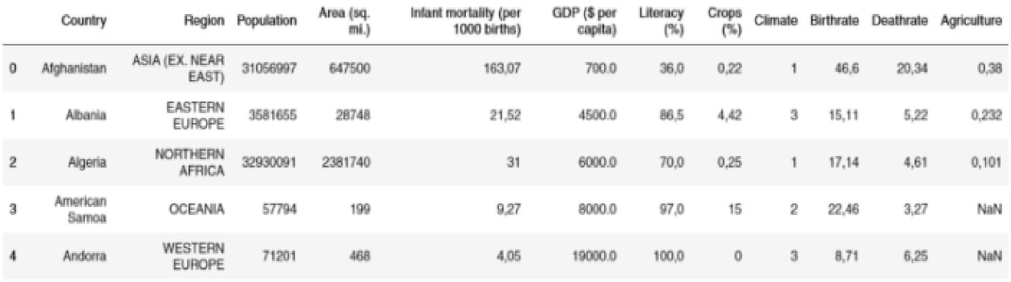
- We will import a csv file by using the **read_csv** method

- **Ex:-**



```
import pandas as pd
countries = pd.read_csv('countries.csv')
countries.head()
```

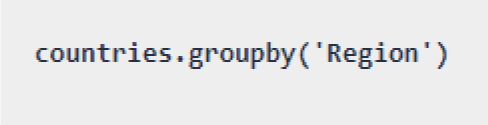
- **Output:-**



	Country	Region	Population	Area (sq. mi.)	Infant mortality (per 1000 births)	GDP ($ per capita)	Literacy (%)	Crops (%)	Climate	Birthrate	Deathrate	Agriculture
0	Afghanistan	ASIA (EX. NEAR EAST)	31056997	647500	163,07	700.0	36,0	0,22	1	46,6	20,34	0,38
1	Albania	EASTERN EUROPE	3581655	28748	21,52	4500.0	86,5	4,42	3	15,11	5,22	0,232
2	Algeria	NORTHERN AFRICA	32930091	2381740	31	6000.0	70,0	0,25	1	17,14	4,61	0,101
3	American Samoa	OCEANIA	57794	199	9,27	8000.0	97,0	15	2	22,46	3,27	NaN
4	Andorra	WESTERN EUROPE	71201	468	4,05	19000.0	100,0	0	3	8,71	6,25	NaN

- Let's say we want to group the dataframe by the region, so we can simply use the **groupby()** method:

- **Ex:-**

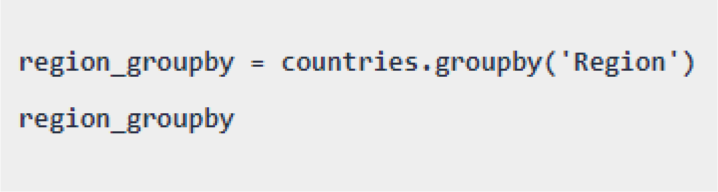


```
countries.groupby('Region')
```

Output:-

- When we apply the ***groupby*** function, a pandas object is returned.
- So in order to work around that, we need to store the grouped dataframe in a variable

- **Ex:-**

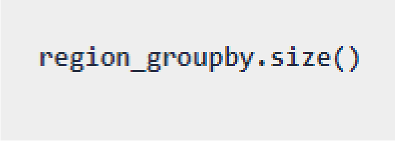


```
region_groupby = countries.groupby('Region')
region_groupby
```

- It still is returned as an object, but now our pandas is stored inside a variable and we can call that variable with different methods as a grouped entity.

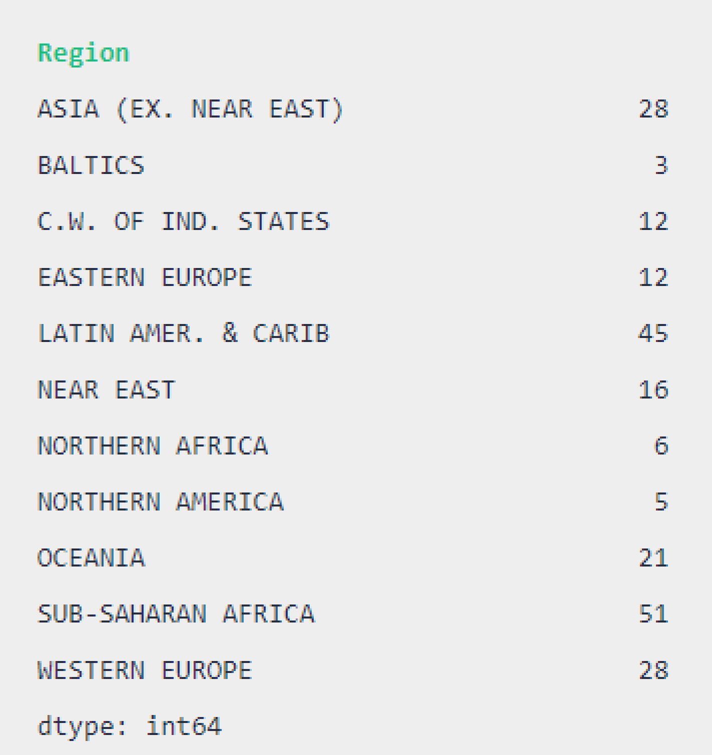
- so let's look at the size of the grouped region dataframe:

- **Ex:-**



```
region_groupby.size()
```

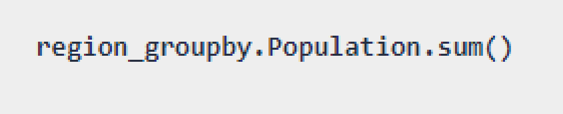
- **Output:-**



```
Region
ASIA (EX. NEAR EAST)             28
BALTICS                           3
C.W. OF IND. STATES              12
EASTERN EUROPE                   12
LATIN AMER. & CARIB              45
NEAR EAST                        16
NORTHERN AFRICA                   6
NORTHERN AMERICA                  5
OCEANIA                          21
SUB-SAHARAN AFRICA               51
WESTERN EUROPE                   28
dtype: int64
```

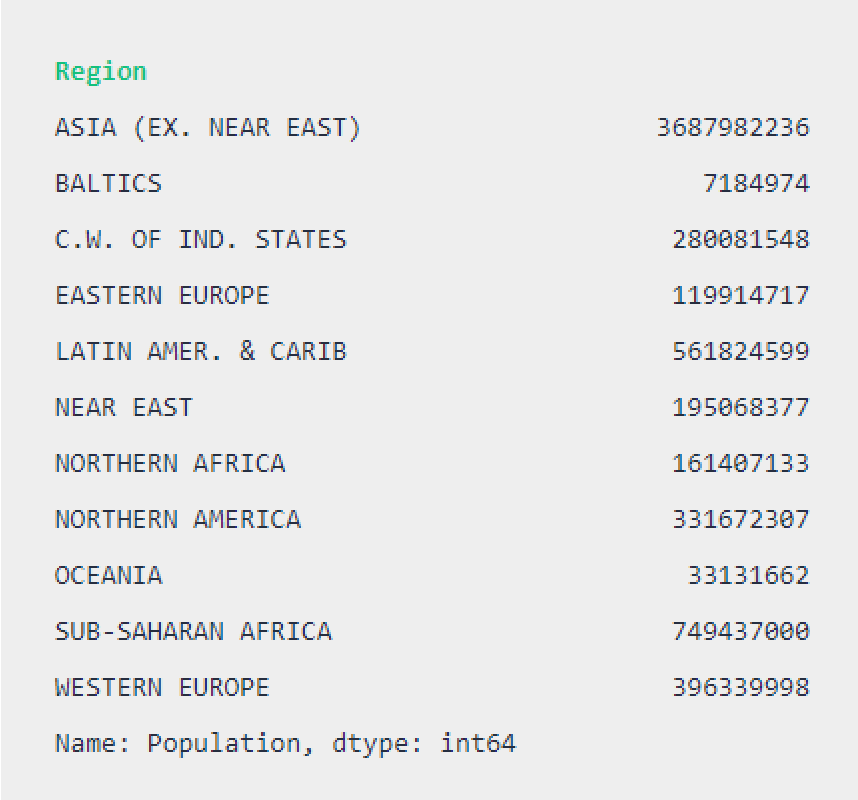
Let's take out the population sum of distributed region area:

- **Ex:-**



```
region_groupby.Population.sum()
```

- **Output:-**

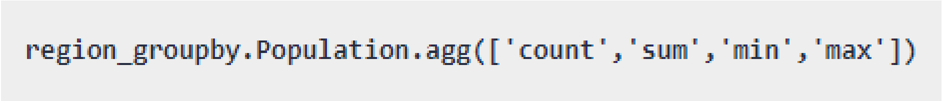


```
Region
ASIA (EX. NEAR EAST)                  3687982236
BALTICS                                  7184974
C.W. OF IND. STATES                    280081548
EASTERN EUROPE                         119914717
LATIN AMER. & CARIB                    561824599
NEAR EAST                              195068377
NORTHERN AFRICA                        161407133
NORTHERN AMERICA                       331672307
OCEANIA                                 33131662
SUB-SAHARAN AFRICA                     749437000
WESTERN EUROPE                         396339998
Name: Population, dtype: int64
```

-

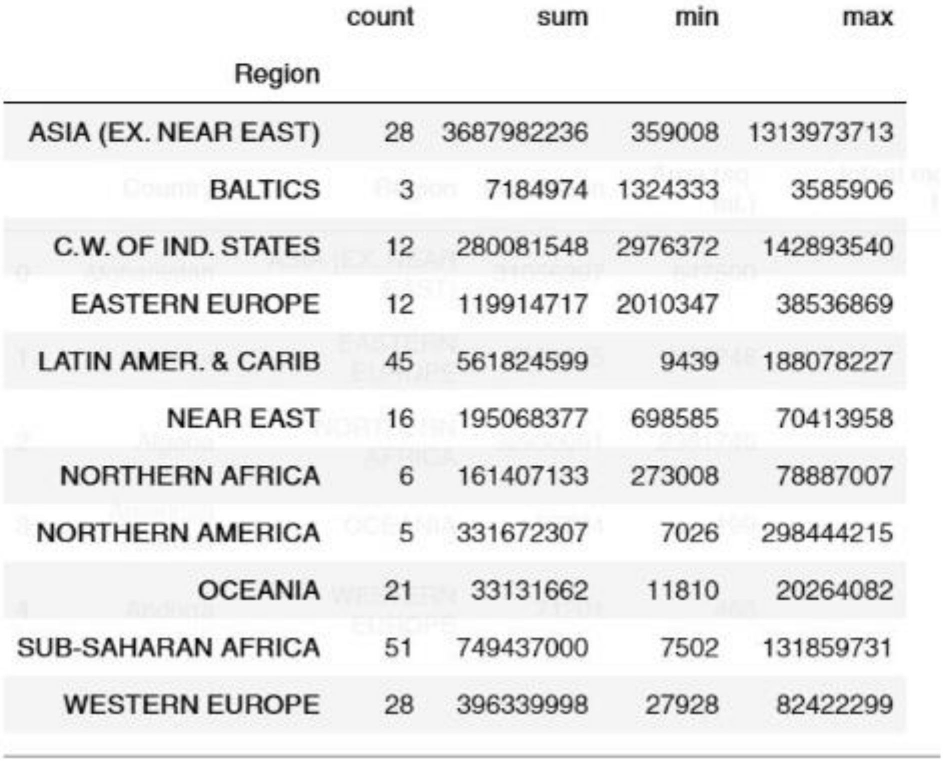
- **You can apply the aggregation function on the population over the region category:**

- **Ex:-**



```
region_groupby.Population.agg(['count','sum','min','max'])
```

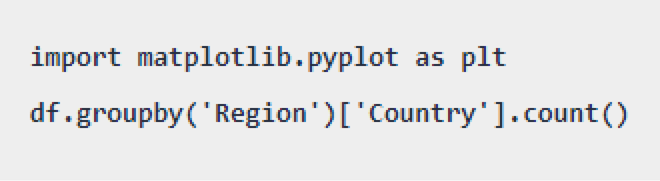
- **Output:-**



	count	sum	min	max
Region				
ASIA (EX. NEAR EAST)	28	3687982236	359008	1313973713
BALTICS	3	7184974	1324333	3585906
C.W. OF IND. STATES	12	280081548	2976372	142893540
EASTERN EUROPE	12	119914717	2010347	38536869
LATIN AMER. & CARIB	45	561824599	9439	188078227
NEAR EAST	16	195068377	698585	70413958
NORTHERN AFRICA	6	161407133	273008	78887007
NORTHERN AMERICA	5	331672307	7026	298444215
OCEANIA	21	33131662	11810	20264082
SUB-SAHARAN AFRICA	51	749437000	7502	131859731
WESTERN EUROPE	28	396339998	27928	82422299

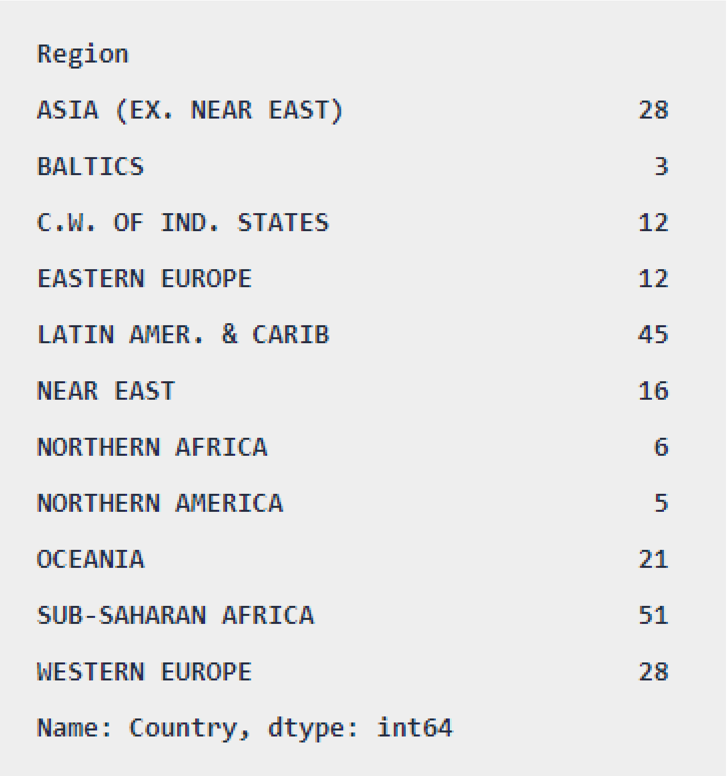
Groupby in Pandas: Plotting with Matplotlib

- You can create a visual display as well to make your analysis look more meaningful by importing matplotlib library. For example, you want to know the number of **Countries** present in each **Region.**
- **Ex:-**



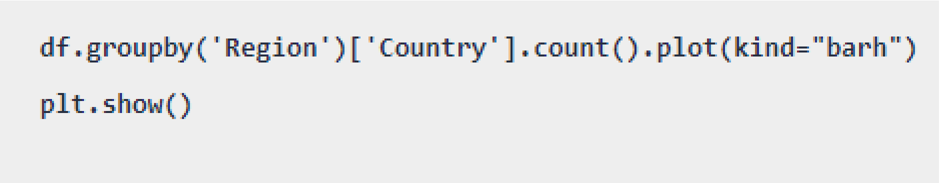
```
import matplotlib.pyplot as plt

df.groupby('Region')['Country'].count()
```

- **Output:-**



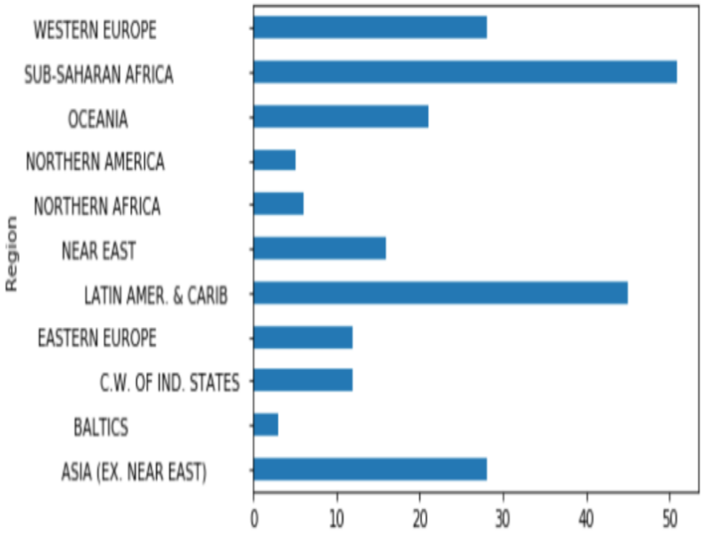
```
Region

ASIA (EX. NEAR EAST)              28

BALTICS                            3

C.W. OF IND. STATES               12

EASTERN EUROPE                    12

LATIN AMER. & CARIB               45

NEAR EAST                         16

NORTHERN AFRICA                    6

NORTHERN AMERICA                   5

OCEANIA                           21

SUB-SAHARAN AFRICA                51

WESTERN EUROPE                    28

Name: Country, dtype: int64
```

- **Let's plot the result now:**
- **Ex:-**



```
df.groupby('Region')['Country'].count().plot(kind="barh")

plt.show()
```

- **Output:-**



What is Missing Data in Pandas?

- Sometimes, you may receive data in bulk which may include missing values or unknown values in rows or columns.
- Handling missing values could be a major task in pandas as you have to necessarily deal with it before applying any algorithm to machine learning otherwise your code won't execute properly.
- So, in order to eliminate the risk of running a bad code, let's learn two different ways of dealing with the missing or unknown values in Pandas:
-

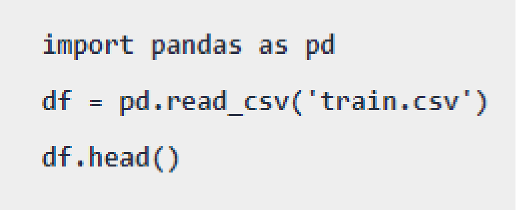
  1. **dropna() method**
  2. **fillna() method**

dropna() Method: Missing Data in Pandas

- Let's work with a dataset called titanic.
- Now, let's import the csv file in order to catch missing values or Nan values.

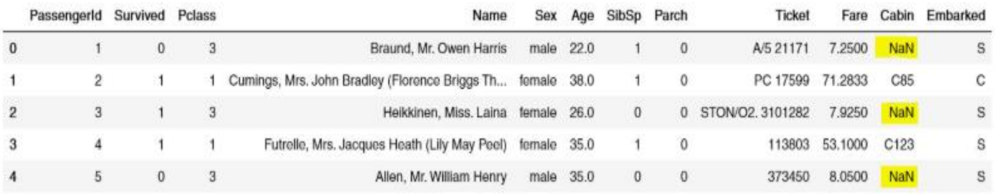
Note:

- NaN values in python stands for missing numerical data, the other representation of **NaN** is **Not a Number**.
- You can also find datasets with values that have None or **Null** in them, it simply means that the cell or container is **empty** or has no value at all.
- **Ex:-**



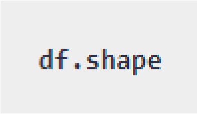
```
import pandas as pd
df = pd.read_csv('train.csv')
df.head()
```

- **Output:-**



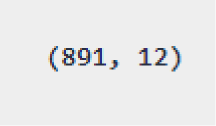
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

-
- You can see the **NaN** values in highlights. Let’s look at the shape of the dataset:
- **Ex:-**



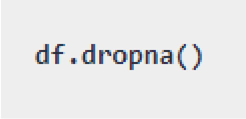
```
df.shape
```

- **Output:-**



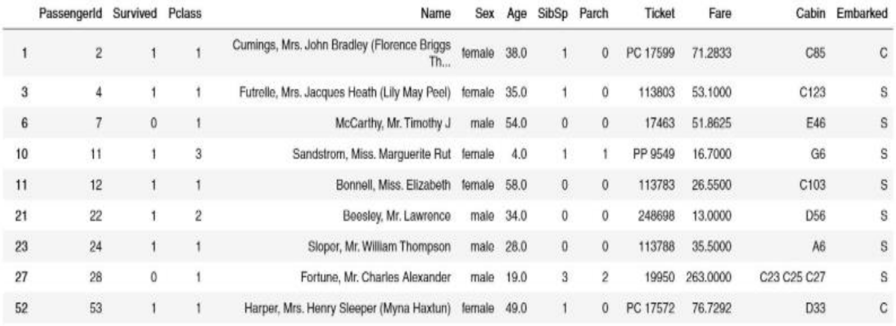
```
(891, 12)
```

- In this particular dataset, we have to deal with a lot of NaN values.
- So let’s learn how to drop such NaN values and clean our dataset.
- **Ex:-**



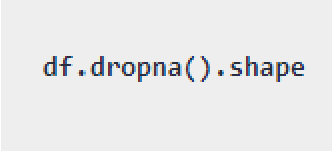
```
df.dropna()
```

- This method will drop the rows which have **NaN** values. So the output will be:



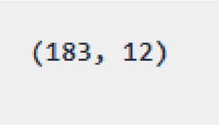
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
10	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.0	1	1	PP 9549	16.7000	G6	S
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S
21	22	1	2	Beesley, Mr. Lawrence	male	34.0	0	0	248698	13.0000	D56	S
23	24	1	1	Sloper, Mr. William Thompson	male	28.0	0	0	113788	35.5000	A6	S
27	28	0	1	Fortune, Mr. Charles Alexander	male	19.0	3	2	19950	263.0000	C23 C25 C27	S
52	53	1	1	Harper, Mrs. Henry Sleeper (Myna Haxtun)	female	49.0	1	0	PC 17572	76.7292	D33	C

-
- Let’s look at the shape of dataframe after dropping NaN values:
- **Ex:-**



```
df.dropna().shape
```

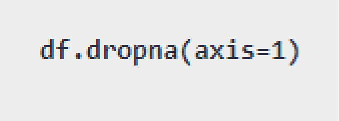
- **Output:-**



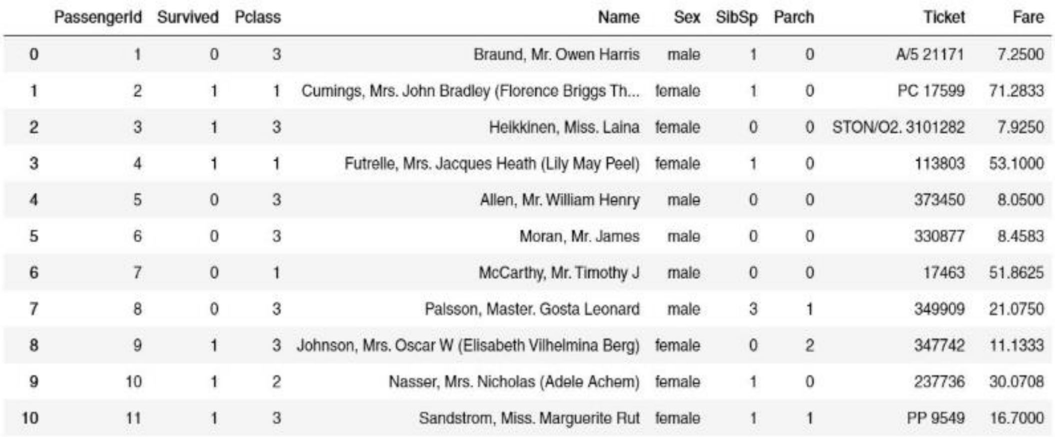
```
(183, 12)
```

- If you want to remove NaN values via columns then you can select the axis set to 1:

- **Ex:-**



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



	PassengerId	Survived	Pclass	Name	Sex	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	1	0	113803	53.1000
4	5	0	3	Allen, Mr. William Henry	male	0	0	373450	8.0500
5	6	0	3	Moran, Mr. James	male	0	0	330877	8.4583
6	7	0	1	McCarthy, Mr. Timothy J	male	0	0	17463	51.8625
7	8	0	3	Palsson, Master. Gosta Leonard	male	3	1	349909	21.0750
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	0	2	347742	11.1333
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	1	0	237736	30.0708
10	11	1	3	Sandstrom, Miss. Marguerite Rut	female	1	1	PP 9549	16.7000

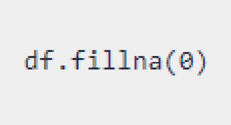
- **Above, you can see that all the columns that had missing values (NaN) are dropped. This is how you can remove or drop the NaN values from your dataset.**

- Note:

- **dropna()** will drop the values temporarily unless you use the in place argument as True to make permanent changes.

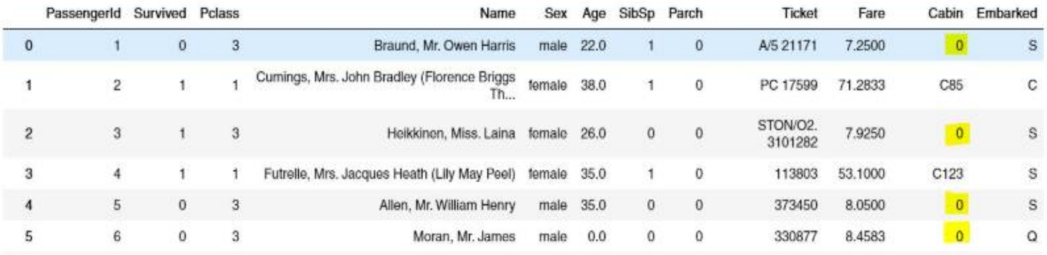
fillna() Method: Missing Data in Pandas

- Now, let’s look at how you can work around missing values without deleting whole rows and columns by filling the voids.
- You can do so by using the **fillna()** method.
- **Ex:-**



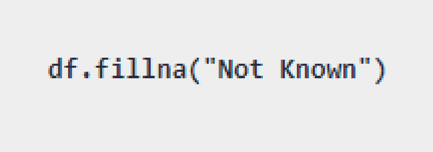
```
df.fillna(0)
```

- **Output:-**



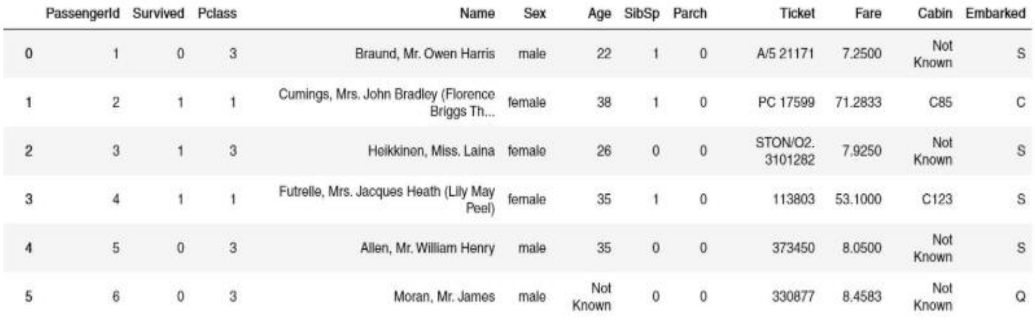
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	0	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	0	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	0	S
5	6	0	3	Moran, Mr. James	male	0.0	0	0	330877	8.4583	0	Q

- You can see that the missing values have been replaced or filled by zeros. Hence, it’s not empty anymore. But sometimes we do come across data that doesn’t have to be always in numbers; hence we need to fill our missing values by strings as well.
- So in order to do that, I can simply put a string inside the fillna() method:
- **Ex:-**



```
df.fillna("Not Known")
```

- **Output:**

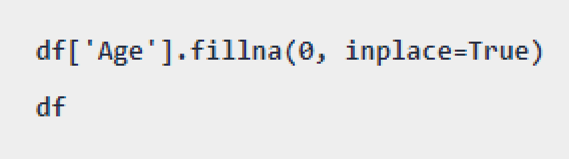


	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.2500	Not Known	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.9250	Not Known	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.0500	Not Known	S
5	6	0	3	Moran, Mr. James	male	Not Known	0	0	330877	8.4583	Not Known	Q

- You can see that the missing value has been replaced with a string “Not Known”, However, this might not be an efficient way of filling missing values as we may encounter a dataframe where we have to replace the missing values by both an integer and a string,

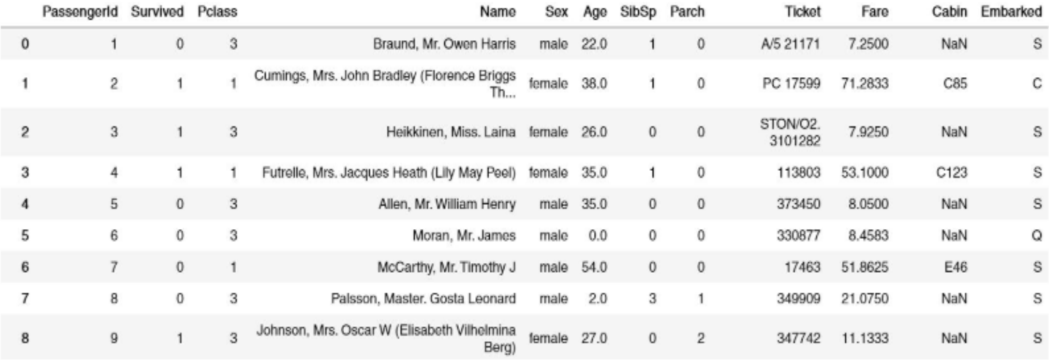
- so we have to select a particularly column to eliminate the confusion.

- **Ex:-**



```
df['Age'].fillna(0, inplace=True)
df
```

- **Output:-**

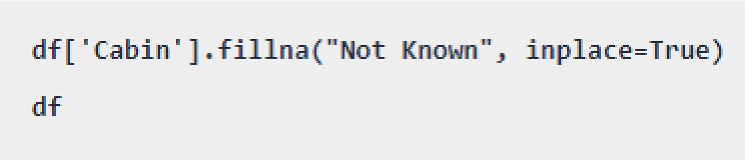


	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	0.0	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	S

- As you can see above, the missing values in the Age column have been replaced with 0.

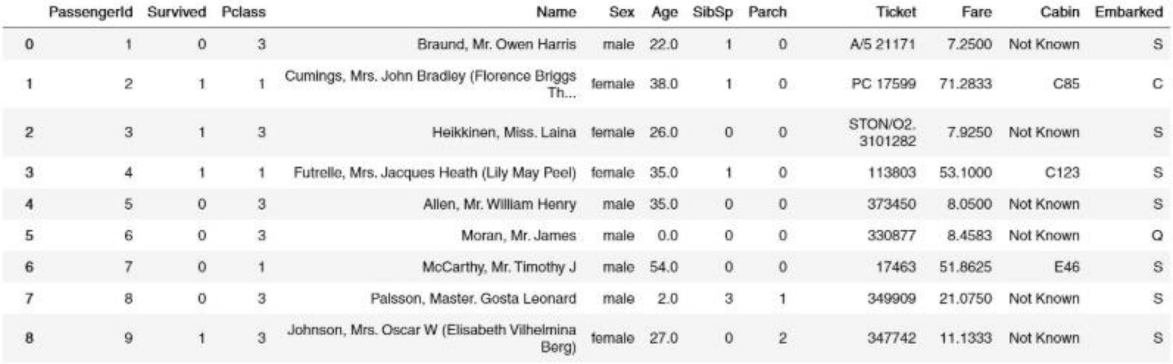
- Similarly, to change the missing values as a string, we can apply the same method with Cabin column as well:

- **EX:-**



```
df['Cabin'].fillna("Not Known", inplace=True)
df
```

- **Output:-**



	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	Not Known	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	Not Known	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	Not Known	S
5	6	0	3	Moran, Mr. James	male	0.0	0	0	330877	8.4583	Not Known	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	Not Known	S
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	Not Known	S

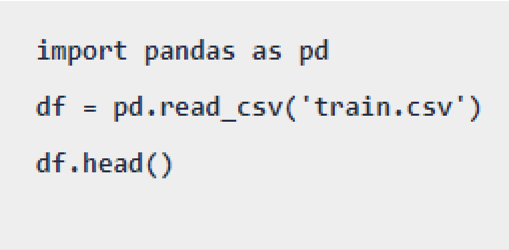
What is a Pivot Table in Pandas?

- If you are familiar with using Microsoft excel then you must be aware of pivot tables as it is the **backbone** for business analysis because it provides a fold of the data provided in new dimensions making data look more summarized and classified.
- We can use the pivot table in Pandas as well using the **pivot_table()** method.

The syntax for pivot_table() is:

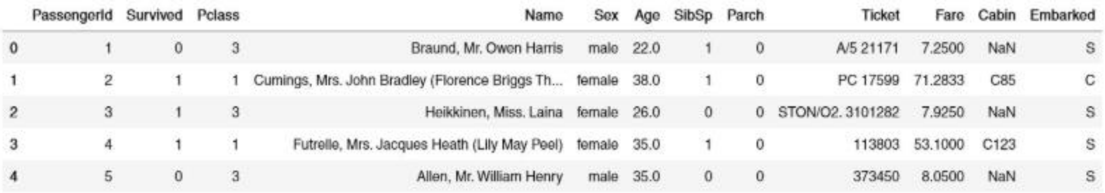
pandas.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All', observed=False) → 'DataFrame'[source]

- **Ex:-**



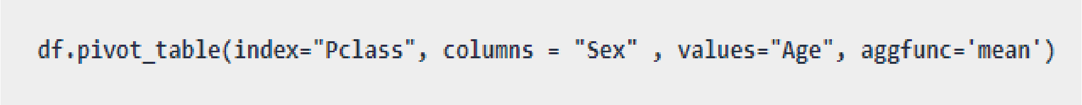
```
import pandas as pd

df = pd.read_csv('train.csv')

df.head()
```

- **Output:-**

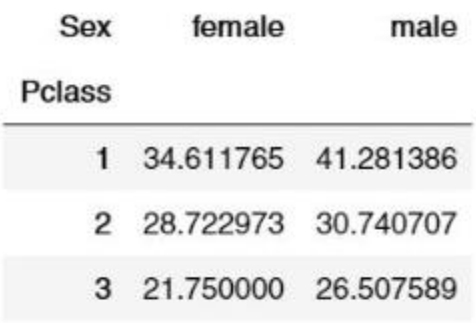


	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

- Let's say that we want to find out the mean age of both male and female based on the **Pclass** they were travelling on, so how do we tell pandas to present us the desired **dataframe**? Well!
- The best way to do that is by using the pivot_table method**.**
- **Ex:-**



```
df.pivot_table(index="Pclass", columns = "Sex" , values="Age", aggfunc='mean')
```



Sex	female	male
Pclass		
1	34.611765	41.281386
2	28.722973	30.740707
3	21.750000	26.507589

Output:-

EXPLANATION:- The above result folds the data based on what we want and displays a new dataframe

Difference Between pivot() and pivot_table() Method:

- pivot() and pivot_table() are two different methods as both of them serve different purposes.
- Main difference between these two methods is:
- **pivot()** is used for pivoting the dataframe without applying aggregation. Hence, it doesn't contain duplicate values or columns/index.
- **pivot_table()** on the other hand will pivot the dataframe by applying aggregation on it, and it will work with managing duplicate values or columns/index

What is Merge in Pandas?

- We have been working with 2-D data which is rows and columns in Pandas.
- In order to go on a higher understanding of what we can do with dataframes that are mostly identical and somehow would join them in order to merge the common values.
- We use a function called **merge()** in pandas that takes the commonalities of two dataframes just like we do in SQL.

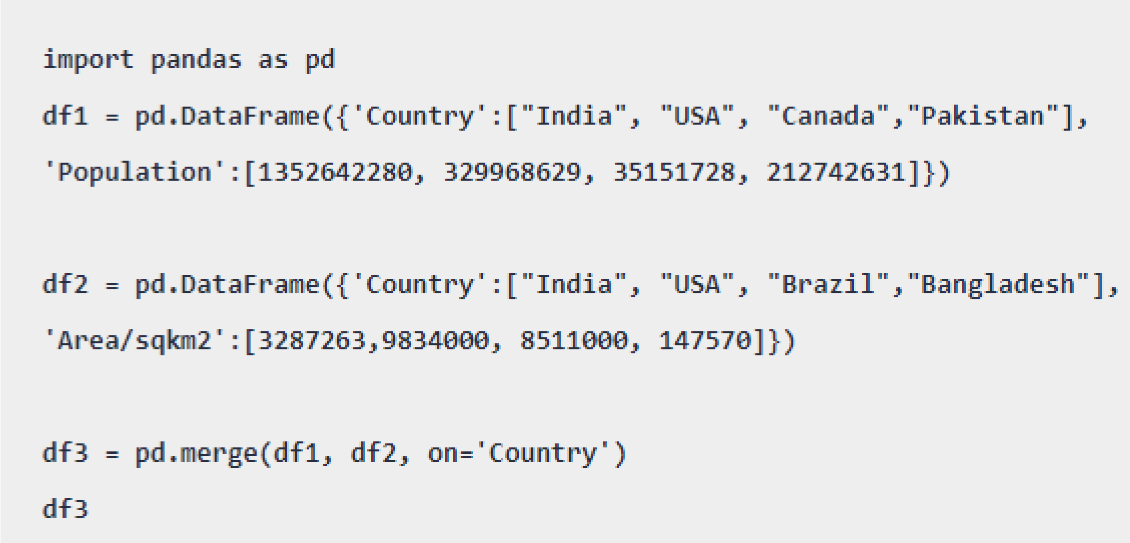
The Syntax for merge in pandas is:

DataFrame.merge(self, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)

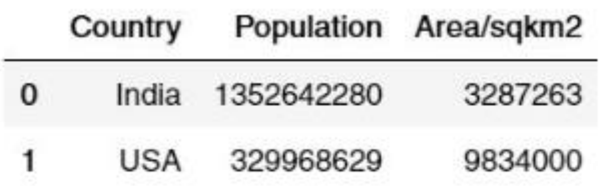
Inner Join

- Let's merge two dataframes which will have common indexes and see different ways to merge their values:

>Ex:-



```
import pandas as pd

df1 = pd.DataFrame({'Country':["India", "USA", "Canada","Pakistan"],

'Population':[1352642280, 329968629, 35151728, 212742631]})


df2 = pd.DataFrame({'Country':["India", "USA", "Brazil","Bangladesh"],

'Area/sqkm2':[3287263,9834000, 8511000, 147570]})


df3 = pd.merge(df1, df2, on='Country')

df3
```

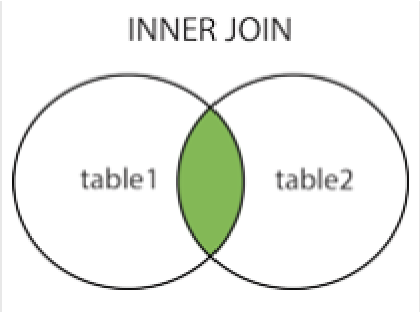


	Country	Population	Area/sqkm2
0	India	1352642280	3287263
1	USA	329968629	9834000

- **Output:-**

  Explanation:-

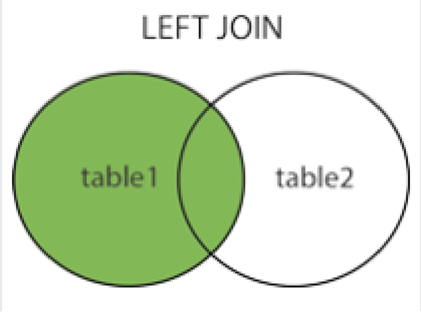
- Above you can see that a simple merge has been displayed which displays the common values present in both the data frames.
- If you want to visually see this, then here is a quick at what exactly happened.



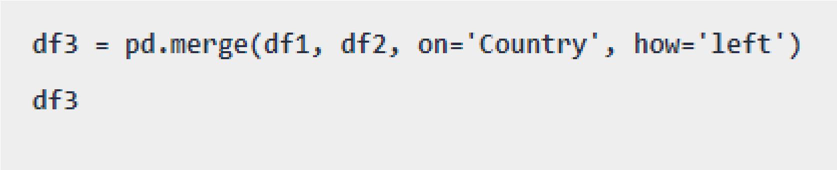
- By default, merge creates the **inner join** which only takes values that are common, in our case **India** and **USA** were present in both the dataframes, so the merge function only printed values that were common in both the dataframes while ignoring other countries. However, we can create, **left** join, **right** join and **outer** join as well by predefining the **'how'** attribute.

Left Merge in Pandas

- As the name suggests the left join will only display values that are common between two dataframes as well as present fully in the left or first dataframe in our case.

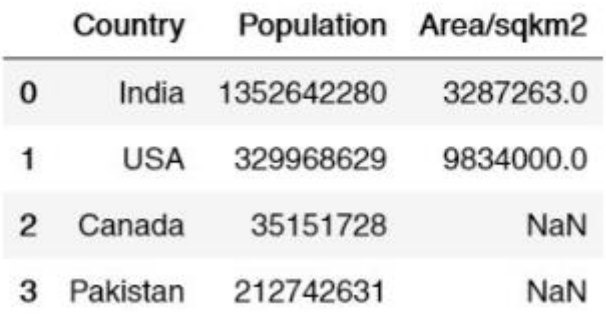


- **Ex:-**



```
df3 = pd.merge(df1, df2, on='Country', how='left')
df3
```

-

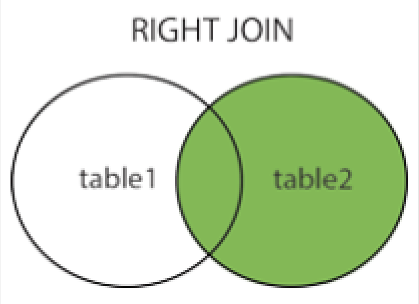


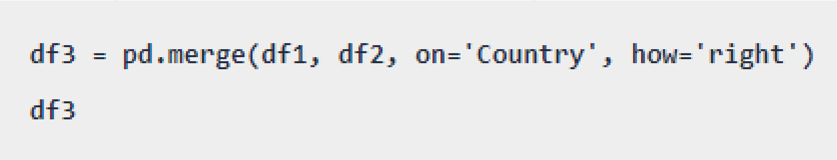
	Country	Population	Area/sqkm2
0	India	1352642280	3287263.0
1	USA	329968629	9834000.0
2	Canada	35151728	NaN
3	Pakistan	212742631	NaN

- **Output:-**
- **Explanation**: You can see that the values that aren’t present or missing during the merge process have been replaced by **NaN** term.

Right Merge in Pandas

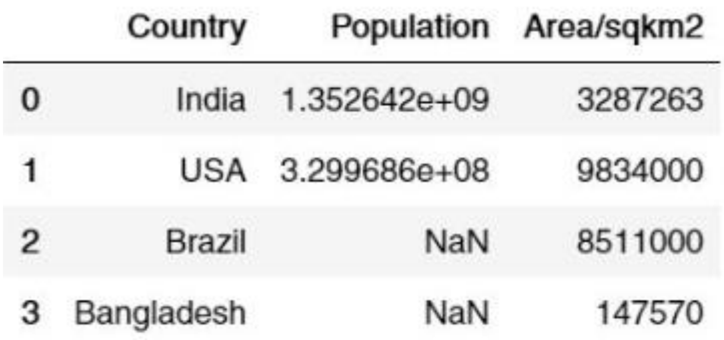
- The left join will only display values that are common between two dataframes as well as present fully in the right dataframe in our case.





```
df3 = pd.merge(df1, df2, on='Country', how='right')
df3
```

- **Ex:-**

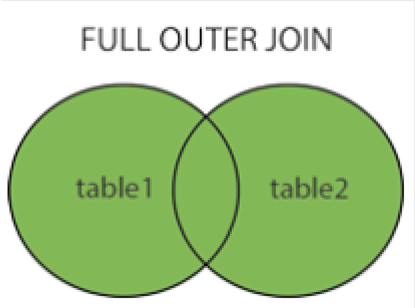


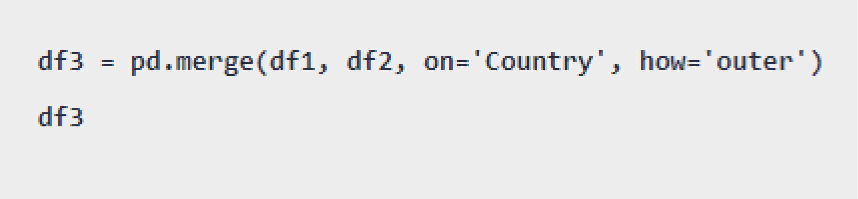
	Country	Population	Area/sqkm2
0	India	1.352642e+09	3287263
1	USA	3.299686e+08	9834000
2	Brazil	NaN	8511000
3	Bangladesh	NaN	147570

- **Output:-**

Outer Join

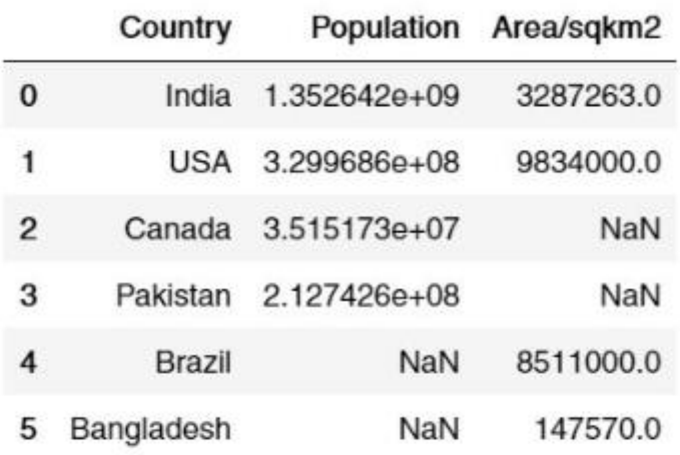
- The outer join will merge all the values together of both the dataframes.





```
df3 = pd.merge(df1, df2, on='Country', how='outer')
df3
```

- Ex:-



	Country	Population	Area/sqkm2
0	India	1.352642e+09	3287263.0
1	USA	3.299686e+08	9834000.0
2	Canada	3.515173e+07	NaN
3	Pakistan	2.127426e+08	NaN
4	Brazil	NaN	8511000.0
5	Bangladesh	NaN	147570.0

- **Output**:-

IMPORTING CSV IN PANDAS:-

- Python is the best choice for performing data analysis mainly because of amazing availability and integration of pandas
- . Pandas is a complete package that can help you import and read data much faster and easier by using a CSV file.
- We can import csv (comma separated values) files by using a method in pandas known as read_csv.
- We need to import csv files because sometimes we might have to work with big size datasets for analysis.
- So, a common format of containing all that data is CSV.

- **The Syntax for read_csv is:**

- **Step 1: Make Sure You Know The Filepath**
- **Step 2: Apply Code to Import CSV in Pandas**
- **Other Steps: You Can Choose Your Own Columns**

The Syntax for read_csv is:

pd.read_csv(filepath_or_buffer, sep=’, ‘, delimiter=None, header=’infer’, names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression=’infer’, thousands=None, decimal=b’.’, lineterminator=None, quotechar=’”‘, quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, doublequote=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None

Step 1: Make Sure You Know The Filepath

- The first important step of importing csv in pandas is to know where the **csv file** is stored.
- It can store on your personal computer or it can be available on the internet in the form of a url with an extension of **‘.csv’**.
- For example on pc, it might be stored as:

C:\Users\Me\Desktop\filename.csv

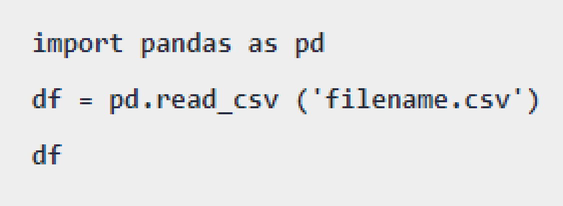
- Firstly, capture the full path where your CSV file is stored. In my case, the CSV file is stored under the following path:

- **C:\Users\Ron\Desktop\Clients.csv**

Step 2: Apply Code to Import CSV in Pandas

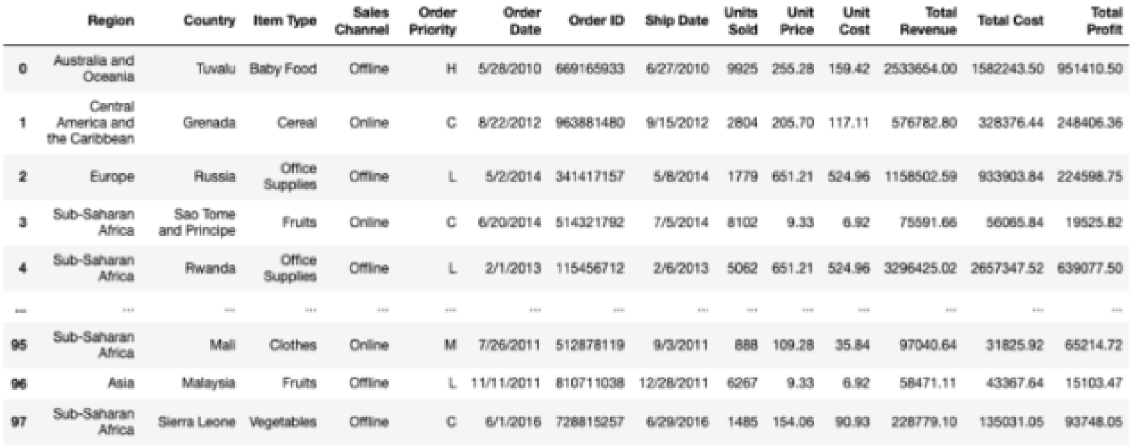
- Import **pandas** in your project and use the **read_csv** function to import the file in a dataframe

- **Ex:-**



```
import pandas as pd

df = pd.read_csv ('filename.csv')

df
```

- **Output:-**



	Region	Country	Item Type	Sales Channel	Order Priority	Order Date	Order ID	Ship Date	Units Sold	Unit Price	Unit Cost	Total Revenue	Total Cost	Total Profit
0	Australia and Oceania	Tuvalu	Baby Food	Offline	H	5/28/2010	669165933	6/27/2010	9925	255.28	159.42	2533654.00	1582243.50	951410.50
1	Central America and the Caribbean	Grenada	Cereal	Online	C	8/22/2012	963881480	9/15/2012	2804	205.70	117.11	576782.80	328376.44	248406.36
2	Europe	Russia	Office Supplies	Offline	L	5/2/2014	341417157	5/8/2014	1779	651.21	524.96	1158502.59	933903.84	224598.75
3	Sub-Saharan Africa	Sao Tome and Principe	Fruits	Online	C	6/20/2014	514321792	7/5/2014	8102	9.33	6.92	75591.66	56065.84	19525.82
4	Sub-Saharan Africa	Rwanda	Office Supplies	Offline	L	2/1/2013	115456712	2/6/2013	5062	651.21	524.96	3296425.02	2657347.52	639077.50
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
95	Sub-Saharan Africa	Mali	Clothes	Online	M	7/26/2011	512878119	9/3/2011	888	109.28	35.84	97040.64	31825.92	65214.72
96	Asia	Malaysia	Fruits	Offline	L	11/11/2011	810711038	12/28/2011	6267	9.33	6.92	58471.11	43367.64	15103.47
97	Sub-Saharan Africa	Sierra Leone	Vegetables	Offline	C	6/1/2016	728815257	6/29/2016	1485	154.06	90.93	228779.10	135031.05	93748.05

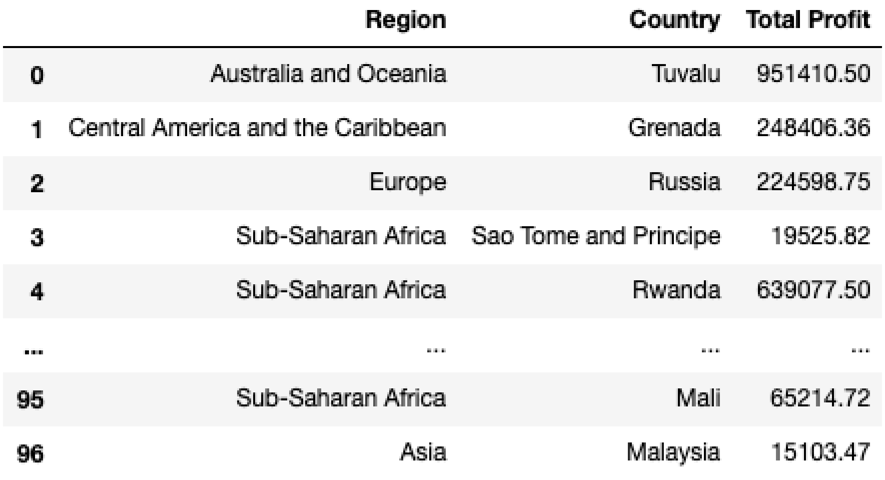
Other Steps: You Can Choose Your Own Columns

- Now let’s say that you want to select a bunch of columns of your own choice within your CSV file when you import it.
- Let’s say that in the sample file we are using; we only need 3 columns for data analysis.
- We can achieve that by using the column attribute.
  - Now what if you want to select a subset of columns from the CSV file?

Ex:-

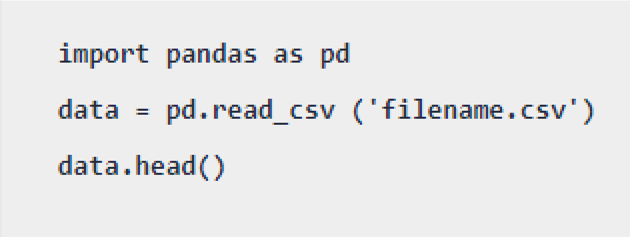
```
import pandas as pd

data = pd.read_csv ('filename.csv')

df = pd.DataFrame(data, columns= ['Region','Country', 'Total Profit'])

df
```

Output:



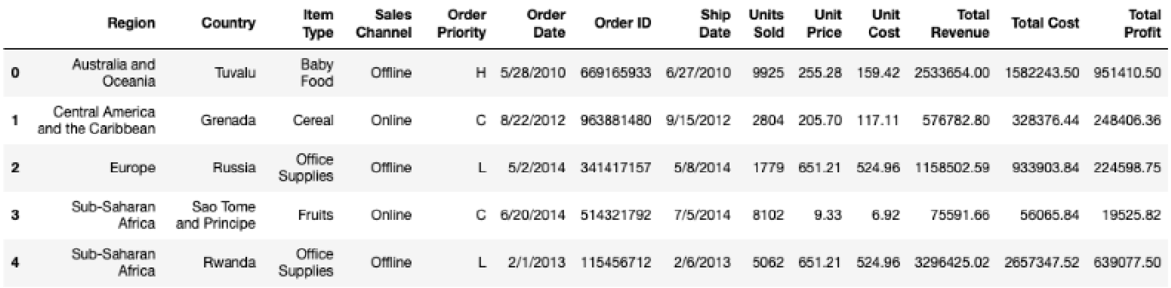
	Region	Country	Total Profit
0	Australia and Oceania	Tuvalu	951410.50
1	Central America and the Caribbean	Grenada	248406.36
2	Europe	Russia	224598.75
3	Sub-Saharan Africa	Sao Tome and Principe	19525.82
4	Sub-Saharan Africa	Rwanda	639077.50
...	...	...	...
95	Sub-Saharan Africa	Mali	65214.72
96	Asia	Malaysia	15103.47

- You can apply the **head()** function as well in order to print the first 5 rows of your dataset from the csv file.



```
import pandas as pd

data = pd.read_csv ('filename.csv')

data.head()
```

- **Ex:-**
- **Output:-**



	Region	Country	Item Type	Sales Channel	Order Priority	Order Date	Order ID	Ship Date	Units Sold	Unit Price	Unit Cost	Total Revenue	Total Cost	Total Profit
0	Australia and Oceania	Tuvalu	Baby Food	Offline	H	5/28/2010	669165933	6/27/2010	9925	255.28	159.42	2533654.00	1582243.50	951410.50
1	Central America and the Caribbean	Grenada	Cereal	Online	C	8/22/2012	963881480	9/15/2012	2804	205.70	117.11	576782.80	328376.44	248406.36
2	Europe	Russia	Office Supplies	Offline	L	5/2/2014	341417157	5/8/2014	1779	651.21	524.96	1158502.59	933903.84	224598.75
3	Sub-Saharan Africa	Sao Tome and Principe	Fruits	Online	C	6/20/2014	514321792	7/5/2014	8102	9.33	6.92	75591.66	56065.84	19525.82
4	Sub-Saharan Africa	Rwanda	Office Supplies	Offline	L	2/1/2013	115456712	2/6/2013	5062	651.21	524.96	3296425.02	2657347.52	639077.50

PLOTTING IN PANDAS

- we are going to learn about the in-built pandas plotting function which is used for visualizing data in various graphs in pandas with the help of **matplotlib** and a dataframe.

- Plotting in Pandas
- Syntax
- Series Plotting in Pandas
- Series Plotting in Pandas – Area Graph
- Scatter Plotting in Pandas
- Bar Plot
- Pie Plotting in Pandas
- Box Plot

Plotting in Pandas

- We can apply different types of plots in pandas in using the matplotlib library which specializes in visually representing the analyzed data
- Pandas has an inbuilt feature of plot which has a following syntax:

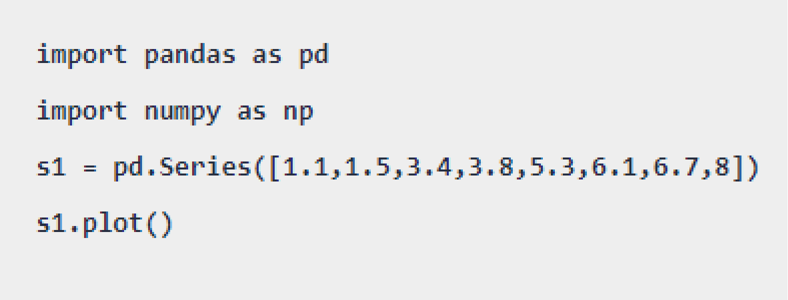
Syntax

-
  ```
  df.plot(
  x=None,
  y=None,
  kind=’line’,
  ax=None,
  subplots=False,
  sharex=None,
  sharey=False,
  layout=None,
  figsize=None,
  use_index=True,
  title=None,
  grid=None,
  legend=True,
  style=None,
  logx=False,
  logy=False,
  loglog=False,
  xticks=None,
  yticks=None,
  xlim=None,
  ylim=None,
  rot=None,
  fontsize=None,
  colormap=None,
  table=False,
  yerr=None,
  xerr=None,
  secondary_y=False,
  sort_columns=False,
  **kwds,
  )
  ```

If you are using jupyter notebook then just import the following libraries to start in Pandas:

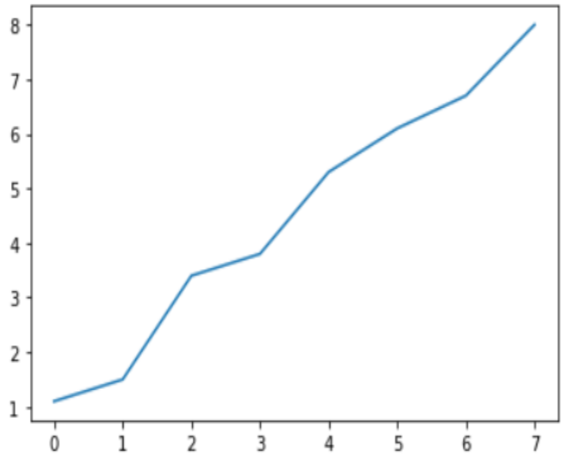
Series Plotting in Pandas

- We can create a whole whole series plot by using the **Series.plot()** method. This type of plot is used when you have a single dimensional data available.
- The example of **Series.plot() is**:



```
import pandas as pd

import numpy as np

s1 = pd.Series([1.1,1.5,3.4,3.8,5.3,6.1,6.7,8])

s1.plot()
```

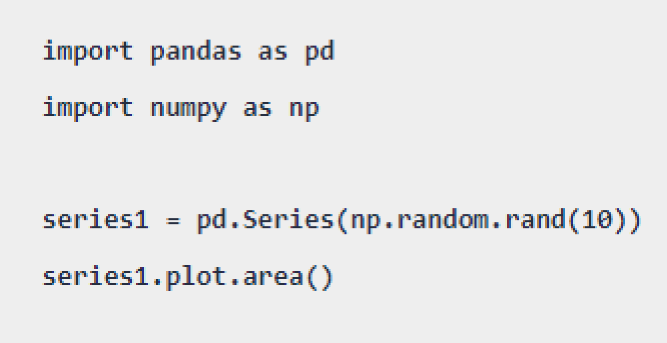
- **Ex:-**



- **Output:-**

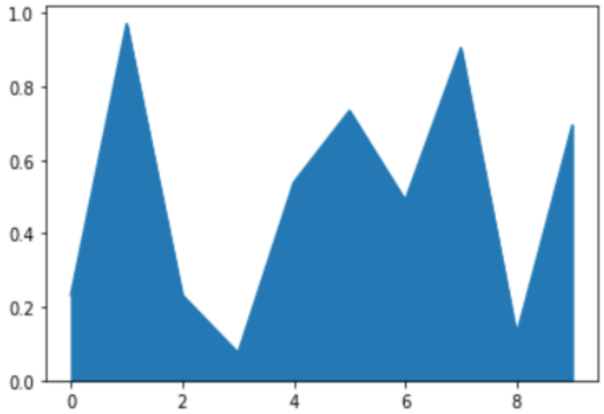
Series Plotting in Pandas – Area Graph

- We can add an area plot in series as well in Pandas using the **Series Plot in Pandas**. This type of series area plot is used for single dimensional data available. > >
  The **example** of series area plot is:



```
import pandas as pd
import numpy as np


series1 = pd.Series(np.random.rand(10))

series1.plot.area()
```

- **Ex:-**

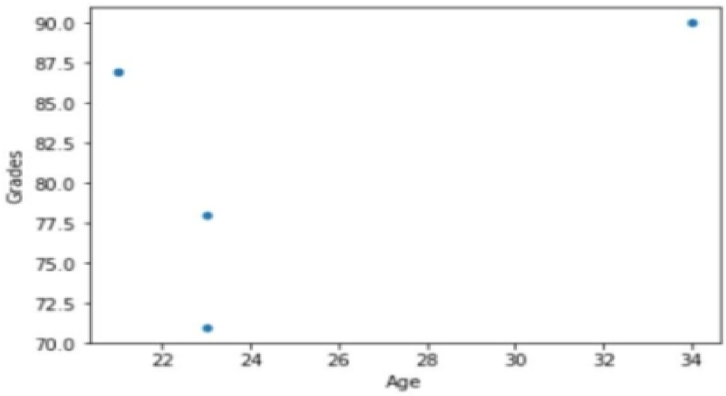


- **Output:-**

Scatter Plotting in Pandas

```
import pandas as pd

import matplotlib.pyplot as plt

import numpy as np


df = pd.DataFrame({'Name':["Hira", "Smith", "Laura","Alex"],

'Age':[23, 34, 21, 23],

'Gender':['f','m','f','m'],

'State':['California','Chicago','Florida','Texas'],

'Grades':[78,90,87,71]})


df.plot(kind='scatter', x='Age', y='Grades')
```

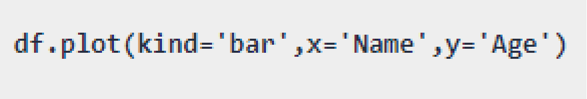
- **Ex:-**



- **Output:-**

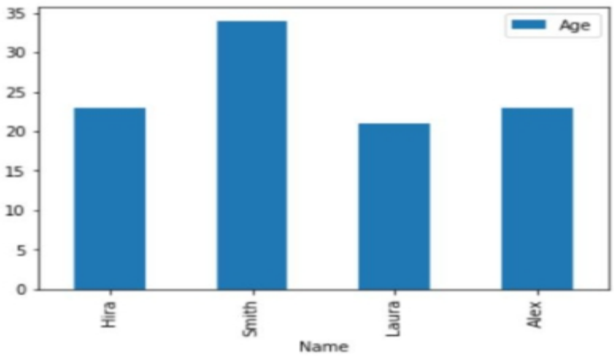
Bar Plot:-

- **Ex:-**



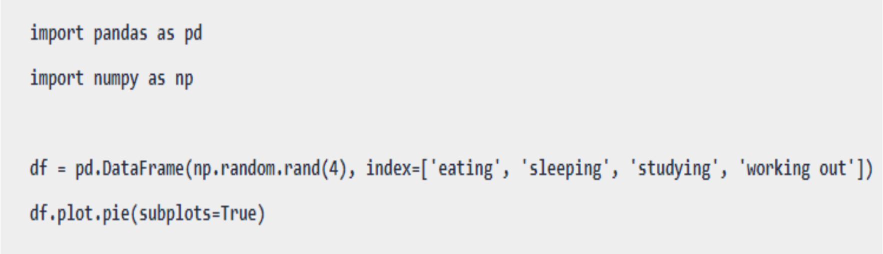
```
df.plot(kind='bar',x='Name',y='Age')
```

- **Output:-**



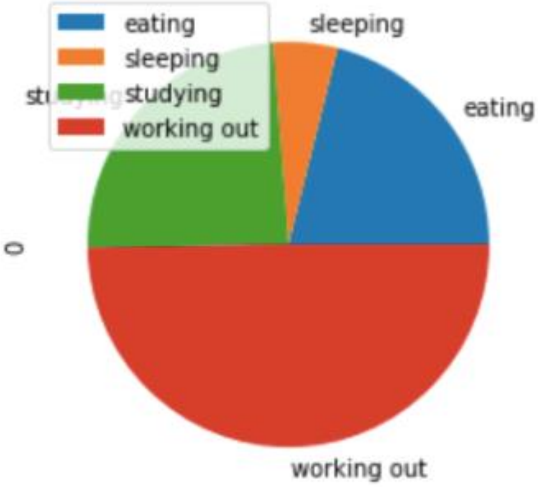
Pie Plotting in Pandas:-

- Pie plot is used for displaying portions or slices of data inside a circle.
- We are able to achieve that by using the matplotlib function known as **dataframe.plot.pie()** for a particular column.
- If no column name is provided then we use the **subplot=True** attribute to draw each numerical data on its own.



```
import pandas as pd
import numpy as np


df = pd.DataFrame(np.random.rand(4), index=['eating', 'sleeping', 'studying', 'working out'])
df.plot.pie(subplots=True)
```

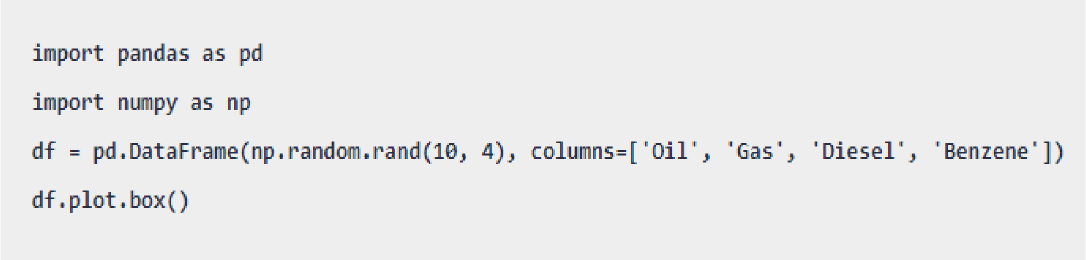
- **Ex:-**



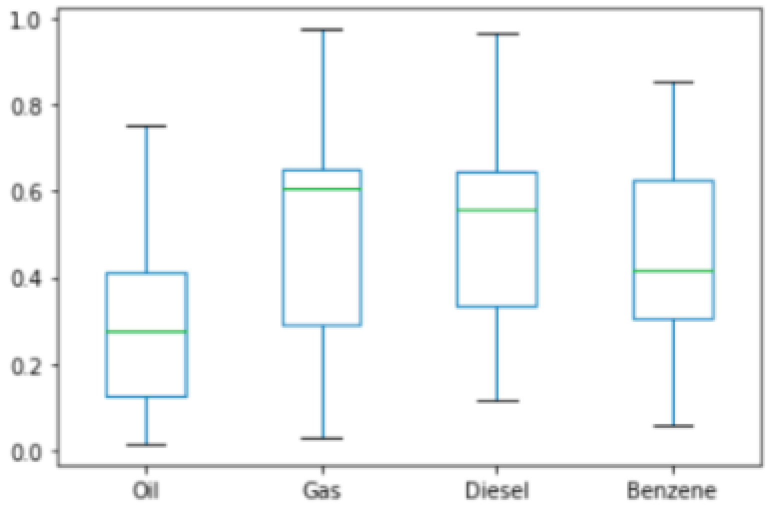
- **Output:**

Box Plot

- A box plot is a way of visually representing different groups of numerical data in quartiles.
- The box starts from Q1 until Q3 quartile and analyses the values with a middle line which is used for calculating median.
- The whiskers at both the end of the box are there to present the data range. Outliers are the points that are present beyond the whiskers.
- Ex:-



```
import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.rand(10, 4), columns=['Oil', 'Gas', 'Diesel', 'Benzene'])

df.plot.box()
```



- **Output:-**