# Fundamentals of Data Science

**WEEK 8: Data Analysis using Pandas**

**Prepared by: Krishna Devkota**

## Import conventions

In [1]:
```
import pandas as pd #import the pandas library and aliasing as pd
import numpy as np # importing NumPy too for later use
```

## Pandas Datastructures

### pandas.Series

A pandas Series can be created using the following constructor:

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

#### Parameter and Description

**data** : data takes various forms like ndarray, list, constants

**index** : Index values must be unique and hashable, same length as data. Default **np.arange(n)** if no index is passed.

**dtype** : dtype is for data type. If None, data type will be inferred

**copy** : Copy data. Default False

#### Interactive Help

In [3]:
```
pd.Series?
```

A series can be created using various inputs like:

- Array
- Dict
- Scalar value or constant

### Create an Empty Series

A basic series, which can be created is an Empty Series.

In [4]:
```
s = pd.Series(dtype=float)
print(s)
```

```
Series([], dtype: float64)
```

### Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where n is array length, i.e., **[0,1,2,3…. range(len(array))-1]**.

#### Example

In [5]:
```
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print(s)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to len(data)-1, i.e., 0 to 3.

**Example**

In [6]:
```
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print(s)
```

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

We passed the index values here. Now we can see the customized indexed values in the output.

## Create a Series from dict

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

**Example**

In [7]:
```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print(s)
```

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

Observe – Dictionary keys are used to construct index.

**Example**

In [8]:
```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print(s)
```

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

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

## Create a Series from Scalar

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

In [9]:
```
s = pd.Series(5, index=[0, 1, 2, 3])
print(s)
```

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

## Accessing Data from Series with Position

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

#### Example 1

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

In [10]:
```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

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

```
1
```

#### Example 2

Retrieve the first three elements in the Series. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index)

In [11]:
```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print(s[:3])
```

```
a    1
b    2
c    3
dtype: int64
```

#### Example 3

Retrieve the last three elements.

In [12]:
```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element
print(s[-3:])
```

```
c    3
d    4
e    5
dtype: int64
```

## Retrieve Data Using Label (Index)

A Series is like a fixed-size dict in that you can get and set values by index label.

#### Example 1

Retrieve a single element using index label value.

In [13]:
```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element
print(s['a'])
```

```
1
```

#### Example 2

Retrieve multiple elements using a list of index label values.

In [14]:
```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
print(s[['a','c','d']])
```

```
a    1
c    3
d    4
dtype: int64
```

**Example 3**

If a label is not contained, an exception is raised.

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

#key is absent, gives an error!
print(s['f'])
```

# Python Pandas - DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame:

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

## Structure:

You can think of it as an SQL table or a spreadsheet data representation.

## pandas.DataFrame

A pandas DataFrame can be created using the following constructor:

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

The parameters of the constructor are as follows:

data : data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.

index : For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.

columns : For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.

dtype : Data type of each column.

copy : This command (or whatever it is) is used for copying of data, if the default is False.

# Create DataFrame

A pandas DataFrame can be created using various inputs like:

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

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

**Create an Empty DataFrame**

A basic DataFrame, which can be created is an Empty Dataframe.

**Example**

In [16]:
```
df = pd.DataFrame()
print(df)
```

```
Empty DataFrame
Columns: []
Index: []
```

## Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

**Example**

In [17]:
```
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

```
   0
0  1
1  2
2  3
3  4
4  5
```

**Example 2**

In [18]:
```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df)
```

```
     Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13
```

**Example 3**

In [19]:
```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df)
```

```
     Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13
```

## Create a DataFrame from Dict of ndarrays / Lists

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

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

**Example 1**

In [20]:
```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)
```

```
    Name  Age
0    Tom   28
1   Jack   34
2  Steve   29
3  Ricky   42
```

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

**Example 2**

Let us now create an indexed DataFrame using arrays.

In [21]:
```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print(df)
```

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

**Note** – Observe, the index parameter assigns an index to each row.

## Create a DataFrame from List of Dicts

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

**Example 1**

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

In [22]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print(df)
```

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

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

**Example 2**

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

In [23]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print(df)
```

```
        a   b     c
first   1   2   NaN
second  5  10  20.0
```

**Example 3**

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

In [24]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

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

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

```
        a   b
first   1   2
second  5  10
        a  b1
first   1 NaN
second  5 NaN
```

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

## Create a DataFrame from Dict of Series

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

**Example:**

In [25]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

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

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

**Note** – Observe, for the series one, there is no label ‘d’ passed, but in the result, for the d label, NaN is appended with NaN.

Let us now understand **column selection**, **addition**, and **deletion** through examples.

## Column Selection

We will understand this by selecting a column from the DataFrame.

**Example**

In [26]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df['one'])
```

```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

## Column Addition

We will understand this by adding a new column to an existing data frame.

**Example**

In [27]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

# Adding a new column to an existing DataFrame object with column label by passing new series

print("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print(df)

print("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']

print(df)
```

```
Adding a new column by passing as Series:
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
Adding a new column using the existing columns in DataFrame:
   one  two  three  four
a  1.0    1   10.0  11.0
b  2.0    2   20.0  22.0
c  3.0    3   30.0  33.0
d  NaN    4    NaN   NaN
```

## Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

#### Example

In [28]:
```
# Using the previous DataFrame, we will delete a column
# using del function

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
   'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)
print("Our dataframe is:")
print(df)

# using del function
print("Deleting the first column using DEL function:")
del df['one']
print(df)

# using pop function
print("Deleting another column using POP function:")
df.pop('two')
print(df)
```

```
Our dataframe is:
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
Deleting the first column using DEL function:
   two  three
a    1   10.0
b    2   20.0
c    3   30.0
d    4    NaN
Deleting another column using POP function:
   three
a   10.0
b   20.0
c   30.0
d    NaN
```

## Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

#### Selection by Label

Rows can be selected by passing row label to a **loc** function.

In [29]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df.loc['b'])
```

```
one    2.0
two    2.0
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

#### Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

In [30]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df.iloc[2])
```

```
one    3.0
two    3.0
Name: c, dtype: float64
```

#### Slice Rows

Multiple rows can be selected using ' : ' operator.

In [31]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df[2:4])
```

```
   one  two
c  3.0    3
d  NaN    4
```

#### Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

In [32]:
```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)
print(df)
```

```
   a  b
0  1  2
1  3  4
0  5  6
1  7  8
```

#### Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

In [33]:
```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0
df = df.drop(0)

print(df)
```

```
   a  b
1  3  4
1  7  8
```

In the above example, two rows were dropped because those two contain the same label 0.

## Python Pandas - Basic Functionality

By now, we learnt about the three Pandas DataStructures and how to create them. We will majorly focus on the **DataFrame** objects because of its importance in the real time data processing.

## Series Basic Functionality

**axes** : Returns a list of the row axis labels

**dtype** : Returns the dtype of the object.

**empty** : Returns True if series is empty.

**ndim** : Returns the number of dimensions of the underlying data, by definition 1.

**size** : Returns the number of elements in the underlying data.

**values** : Returns the Series as ndarray.

**head()** : Returns the first n rows.

**tail()** : Returns the last n rows.

In [34]:
```
import pandas as pd
import numpy as np

#Create a series with 100 random numbers
s = pd.Series(np.random.randn(4))
print("The axes are:")
print(s.axes)
```

```
The axes are:
[RangeIndex(start=0, stop=4, step=1)]
```

In [35]:
```
s.dtype
```

Out[35]:
```
dtype('float64')
```

In [36]:
```
s.empty
```

Out[36]:
```
False
```

In [37]:
```
s.ndim
```

Out[37]:
```
1
```

In [38]:
```
s.size
```

Out[38]:
```
4
```

In [39]:
```
s.values
```

Out[39]:
```
array([ 0.68345806, -1.51230528, -1.3507157 ,  1.11540978])
```

In [40]:
```
1 s.head()
```

Out[40]:
```
0    0.683458
1   -1.512305
2   -1.350716
3    1.115410
dtype: float64
```

In [41]:
```
1 s.tail()
```

Out[41]:
```
0    0.683458
1   -1.512305
2   -1.350716
3    1.115410
dtype: float64
```

## DataFrame Basic Functionality

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

**T** : Transposes rows and columns.

**axes** : Returns a list with the row axis labels and column axis labels as the only members.

**dtypes** : Returns the dtypes in this object.

**empty** : True if NDFrame is entirely empty [no items]; if any of the axes are of length 0.

**ndim** : Number of axes / array dimensions.

**shape** : Returns a tuple representing the dimensionality of the DataFrame.

**size** : Number of elements in the NDFrame.

**values** : Numpy representation of NDFrame.

**head()** : Returns the first n rows.

**tail()** : Returns last n rows.

Let us now create a DataFrame and see all how the above mentioned attributes operate.

In [42]:
```
1 #Create a Dictionary of series
2 d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
3    'Age':pd.Series([25,26,25,23,30,29,23]),
4    'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}
5
6 #Create a DataFrame
7 df = pd.DataFrame(d)
8 print("Our data series is:")
9 print(df)
```

```
Our data series is:
    Name  Age  Rating
0    Tom   25    4.23
1  James   26    3.24
2  Ricky   25    3.98
3    Vin   23    2.56
4  Steve   30    3.20
5  Smith   29    4.60
6   Jack   23    3.80
```

In [43]:
```
1 df.T
```

Out[43]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Name** | Tom | James | Ricky | Vin | Steve | Smith | Jack |
| **Age** | 25 | 26 | 25 | 23 | 30 | 29 | 23 |
| **Rating** | 4.23 | 3.24 | 3.98 | 2.56 | 3.2 | 4.6 | 3.8 |

In [44]:
```
1 df.axes
```

Out[44]:
```
[RangeIndex(start=0, stop=7, step=1),
 Index(['Name', 'Age', 'Rating'], dtype='object')]
```

In [45]:
```
df.dtypes
```

Out[45]:
```
Name       object
Age         int64
Rating    float64
dtype: object
```

In [46]:
```
df.ndim
```

Out[46]:
```
2
```

In [47]:
```
df.shape
```

Out[47]:
```
(7, 3)
```

In [48]:
```
df.size
```

Out[48]:
```
21
```

In [49]:
```
df.values
```

Out[49]:
```
array([['Tom', 25, 4.23],
       ['James', 26, 3.24],
       ['Ricky', 25, 3.98],
       ['Vin', 23, 2.56],
       ['Steve', 30, 3.2],
       ['Smith', 29, 4.6],
       ['Jack', 23, 3.8]], dtype=object)
```

In [50]:
```
df.head(2)
```

Out[50]:

| | Name | Age | Rating |
|---|---|---|---|
| **0** | Tom | 25 | 4.23 |
| **1** | James | 26 | 3.24 |

In [51]:
```
df.tail()
```

Out[51]:

| | Name | Age | Rating |
|---|---|---|---|
| **2** | Ricky | 25 | 3.98 |
| **3** | Vin | 23 | 2.56 |
| **4** | Steve | 30 | 3.20 |
| **5** | Smith | 29 | 4.60 |
| **6** | Jack | 23 | 3.80 |

## Python Pandas - Descriptive Statistics

A large number of methods collectively compute descriptive statistics and other related operations on DataFrame.

Most of these are aggregations like sum(), mean(), but some of them, like sumsum(), produce an object of the same size.

Generally speaking, these methods take an axis argument, just like ndarray.{sum, std, ...}, but the axis can be specified by name or integer.

```
DataFrame – “index” (axis=0, default), “columns” (axis=1)
```

**Example**

In [52]:
```
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print(df)
```

```
      Name  Age  Rating
0      Tom   25    4.23
1    James   26    3.24
2    Ricky   25    3.98
3      Vin   23    2.56
4    Steve   30    3.20
5    Smith   29    4.60
6     Jack   23    3.80
7      Lee   34    3.78
8    David   40    2.98
9   Gasper   30    4.80
10  Betina   51    4.10
11  Andres   46    3.65
```

**sum( )** : Returns the sum of the values for the requested axis. By default, axis is index (axis=0).

In [53]:
```
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print(df.sum())
```

```
Name      TomJamesRickyVinSteveSmithJackLeeDavidGasperBe...
Age                                                     382
Rating                                                44.92
dtype: object
```

Each individual column is added individually (Strings are appended).

axis=1 : This syntax will give the output as shown below.

In [54]:
```
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print(df[['Age','Rating']].sum(1))
```

```
0     29.23
1     29.24
2     28.98
3     25.56
4     33.20
5     33.60
6     26.80
7     37.78
8     42.98
9     34.80
10    55.10
11    49.65
dtype: float64
```

In [55]:
```
df[['Age','Rating']].mean()
```

Out[55]:
```
Age       31.833333
Rating     3.743333
dtype: float64
```

In [56]:
```
1 df[['Age','Rating']].mean(1)
```

Out[56]:
```
0     14.615
1     14.620
2     14.490
3     12.780
4     16.600
5     16.800
6     13.400
7     18.890
8     21.490
9     17.400
10    27.550
11    24.825
dtype: float64
```

In [57]:
```
1 df[['Age','Rating']].std()
```

Out[57]:
```
Age       9.232682
Rating    0.661628
dtype: float64
```

## Functions & Description

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

**count()** : Number of non-null observations

**sum()** : Sum of values

**mean()** : Mean of Values

**median()** : Median of Values

**mode()** : Mode of values

**std()** : Standard Deviation of the Values

**min()** : Minimum Value

**max()** : Maximum Value

**abs()** : Absolute Value

**prod()** : Product of Values

**cumsum()** : Cumulative Sum

**cumprod()** : Cumulative Product

**Note** – Since DataFrame is a Heterogeneous data structure. Generic operations don't work with all functions.

Functions like **sum()**, **cumsum()** work with both numeric and character (or) string data elements without any error. Though n practice, character aggregations are never used generally, these functions do not throw any exception.

Functions like **abs()**, **cumprod()** throw exception when the DataFrame contains character or string data because such operations cannot be performed.

## Summarizing Data

The **describe( )** function computes a summary of statistics pertaining to the DataFrame columns.

In [58]:
```
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print(df.describe())
```

```
             Age     Rating
count  12.000000  12.000000
mean   31.833333   3.743333
std     9.232682   0.661628
min    23.000000   2.560000
25%    25.000000   3.230000
50%    29.500000   3.790000
75%    35.500000   4.132500
max    51.000000   4.800000
```

This function gives the **mean**, **std** and **IQR** values. And, function excludes the character columns and given summary about numeric columns. **'include'** is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.

**object** – Summarizes String columns

**number** – Summarizes Numeric columns

**all** – Summarizes all columns together (Should not pass it as a list value)

Now, use the following statement in the program and check the output:

In [59]:
```
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print(df.describe(include=['object']))
```

```
       Name
count    12
unique   12
top     Tom
freq      1
```

Now, use the following statement and check the output:

In [60]:
```
#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])
}

#Create a DataFrame
df = pd.DataFrame(d)
print(df. describe(include='all'))
```

```
       Name        Age     Rating
count    12  12.000000  12.000000
unique   12        NaN        NaN
top     Tom        NaN        NaN
freq      1        NaN        NaN
mean    NaN  31.833333   3.743333
std     NaN   9.232682   0.661628
min     NaN  23.000000   2.560000
25%     NaN  25.000000   3.230000
50%     NaN  29.500000   3.790000
75%     NaN  35.500000   4.132500
max     NaN  51.000000   4.800000
```

## Python Pandas - IO Tools

The Pandas I/O API is a set of top level reader functions accessed like **pd.read_csv( )** that generally return a Pandas object.

The two workhorse functions for reading text files (or the flat files) are **read_csv( )** and **read_table( )**. They both use the same parsing code to intelligently convert tabular data into a DataFrame object

In [61]:
```
import pandas as pd

df=pd.read_csv("data\weight_height_data.csv")
print(df)
```

```
     weight  height  gender
0     64.31  156.69    male
1     47.18  158.88    male
2     47.21  162.66    male
3     59.14  158.42    male
4     65.14  161.60    male
..      ...     ...     ...
995   56.50  156.25  female
996   45.82  151.24  female
997   54.99  154.10  female
998   43.62  150.53  female
999   57.21  156.16  female

[1000 rows x 3 columns]
```