# PYTHON PANDAS

## Series and Data Frames Starter

Rishi
Rishi's Python Starter Course

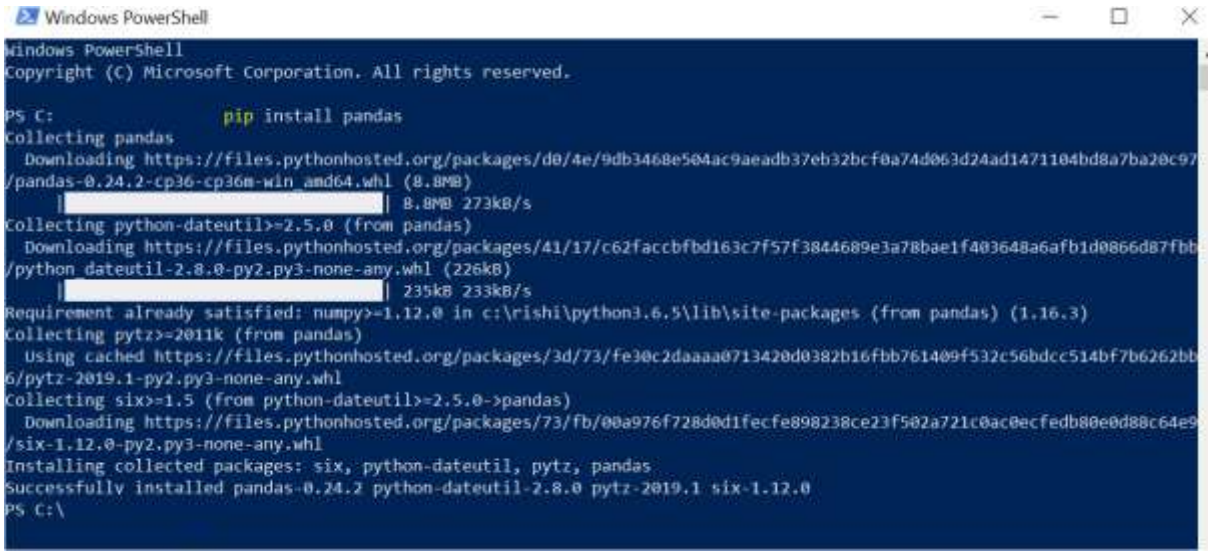


rishi.arun@yahoo.com

Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Python with In this tutorial, we will learn the various features of Python Pandas and how to use them in practice.

To install Pandas, use PIP command as follows:

**pip install pandas**




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

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

**Key Features of Pandas**

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

**Introduction to Data Structures**

Pandas deals with the following three data structures −

- Series
- DataFrame

- Panel - Deprecated

These data structures are built on top of Numpy array, which means they are fast. The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, **DataFrame** is a container of **Series**, **Panel** is a container of DataFrame.

| Data Structure | Dimensions | Description |
|---|---|---|
| Series | 1 | 1D labelled homogeneous array, size immutable. |
| Data Frames | 2 | General 2D labelled, size-mutable tabular structure with potentially heterogeneously typed columns. |
| Panel | 3 | General 3D labelled, size-mutable array. |

**Mutability**

All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.

**Note** – DataFrame is widely used and one of the most important data structures. Panel is used much less.

# Series

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

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 23 | 56 | 17 | 52 | 61 | 73 | 90 | 26 | 72 |

**Key Points**

- Homogeneous data
- Size Immutable
- Values of Data Mutable

# DataFrame

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

| Name | Age | Gender | Rating |
|---|---|---|---|
| Steve | 32 | Male | 3.45 |
| Lia | 28 | Female | 4.6 |

| | | | |
|---|---|---|---|
| Vin | 45 | Male | 3.9 |
| Katie | 38 | Female | 2.78 |

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

**Key Points**

- Homogeneous data
- Size Immutable
- Data Mutable

# pandas.Series

A pandas Series can be created using the following constructor –

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print(s)
```

Its output is as follows –

```
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].

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

Its output is as follows –

```
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.

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

Its output is as follows –

```
100  a
101  b
```

```
102  c
103  d
dtype: object
```

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

```
import pandas as pd
import numpy as np

data = np.array(['a','b','c','d'])
index_list = np.arange(1, data.size + 1)
s = pd.Series(data, index=index_list)
print(data)
print(s)
```

**Create a Series from dictionary:**

A dictionary 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.

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

Its output is as follows −

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

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

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

Its output is as follows −

```
b 1.0
```

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

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

### Create a Series from Scalar

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

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

Its output is as follows −

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

## Accessing Data from Series with Position

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

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.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve the first element
print(s[0])
```

Its output is as follows −

```
1
```

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)

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve the first three element
print (s[:3])
```

Its output is as follows −

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

**Retrieve the last three elements.**

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve the last three element
print (s[-3:])
```

Its output is as follows −

```
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.
Retrieve a single element using index label value.

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

Its output is as follows −

```
1
```

Retrieve multiple elements using a list of index label values.

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

Its output is as follows −

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

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

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

```
print (s['f'])
```

Its output is as follows −

**KeyError**: 'f'

# 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.

Let us now create a Series and see all the above tabulated attributes operation.

```
import pandas as pd
import numpy as np

#Create a series with 25 random numbers
s = pd.Series(np.random.randn(10))
print (s)
print ("The axes are:")
print (s.axes)
```

Returns the Boolean value saying whether the Object is empty or not. True indicates that the object is empty.

```
print ("Is the Object empty?")
print (s.empty)
```

### ndim

Returns the number of dimensions of the object. By definition, a Series is a 1D data structure, so it returns

```
print ("The dimensions of the object:")
print (s.ndim)
```

### size

Returns the size(length) of the series.

```
print ("The size of the object:")
print (s.size)
```

### values

Returns the actual data in the series as an array.

```
print ("The actual data series is:")
print (s.values)
```

head() returns the first n rows(observe the index values). The default number of elements to display is five, but you may pass a custom number.

```
print ("The first two rows of the data series:")
print (s.head(2))
```

tail() returns the last n rows(observe the index values). The default number of elements to display is five, but you may pass a custom number.

```
print ("The last two rows of the data series:")
print (s.tail(2))
```

## 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 access (rows and columns)
- Can Perform Arithmetic operations on rows and columns

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*)

**Create DataFrame**

A pandas DataFrame can be created using various inputs like –

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

**Create an Empty DataFrame**

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

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print (df)
```

Its output is as follows –

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

**Create a DataFrame from Lists**

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

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print (df)
```

Its output is as follows −

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

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print (df)
```

Its output is as follows −

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

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print (df)
```

Its output is as follows −

```
     Name     Age
0    Alex    10.0
1    Bob     12.0
2    Clarke  13.0
```

Note − Observe, the dtype parameter changes the type of Age column to floating point.

**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.

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

```
print (df)
```

Its output is as follows −

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

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

**Let us now create an indexed DataFrame using arrays.**

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

Its output is as follows −

```
          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.

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

Its output is as follows −

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

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

Example 2
The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

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

Its output is as follows −

```
   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.

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

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

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print (df)1
print (df)2
```

Its output is as follows −

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

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

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

## Create a DataFrame from Dict of Series

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

Example

```
import pandas as pd

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

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

Its output is as follows −

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

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.

```
import pandas as pd

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

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

Its output is as follows –

```
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.

```
import pandas as pd

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

df = pd.DataFrame(d)

# 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)
```

Its output is as follows −

```
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.

```
# Using the previous DataFrame, we will delete a column
# using del function
import pandas as pd

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

**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.

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

```
import pandas as pd

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

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

Its output is as follows −

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

**Slice Rows**
Multiple rows can be selected using ‘ : ’ operator.

```
import pandas as pd

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

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

Its output is as follows −

```
   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.

```
import pandas as pd

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)
```

Its output is as follows −

```
   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.

```
import pandas as pd

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)
```

Its output is as follows −

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

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

# DataFrame Basic Functionality

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

```
import pandas as pd
import numpy as np

#Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack']),
   'Age':pd.Series([25,26,25,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

#Create a DataFrame
df = pd.DataFrame(d)
print ("Our data series is:")
print (df)
```

Its output is as follows −

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

### *T (Transpose)*

Returns the transpose of the DataFrame. The rows and columns will interchange.

```
print ("The transpose of the data series is:")
print(df.T)
```

### *axes*

Returns the list of row axis labels and column axis labels.

```
print ("Row axis labels and column axis labels are:")
print (df.axes)
```

### *dtypes*

Returns the data type of each column.

```
print ("The data types of each column are:")
```

*print (df.dtypes)*

*empty*

Returns the Boolean value saying whether the Object is empty or not; True indicates that the object is empty.

*print ("Is the object empty?")*
*print (df.empty)*

*ndim*

Returns the number of dimensions of the object. By definition, DataFrame is a 2D object.

*print ("The dimension of the object is:")*
*print (df.ndim)*

*shape*

Returns a tuple representing the dimensionality of the DataFrame. Tuple (a,b), where a represents the number of rows and b represents the number of columns.

*print ("The shape of the object is:")*
*print (df.shape)*

*size*

Returns the number of elements in the DataFrame.

*print ("The total number of elements in our object is:")*
*print(df.size)*

*values*

Returns the actual data in the DataFrame as an NDarray.

*print ("The actual data in our data frame is:")*
*print (df.values)*

*Head & Tail*

To view a small sample of a DataFrame object, use the head() and tail() methods. head() returns the first n rows (observe the index values).
The default number of elements to display is five, but you may pass a custom number.

*print ("The first two rows of the data frame is:")*
*print (df.head(2))*

tail() returns the last n rows (observe the index values). The default number of elements to display is five, but you may pass a custom number.

*print ("The last two rows of the data frame is:")*
*print (df.tail(2))*

# 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)*

Let us create a DataFrame and use this object throughout this chapter for all the operations.

```
import pandas as pd
import numpy as np

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

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

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

```
df = pd.DataFrame(d)
print (df.sum(1))
```

mean()
Returns the average value

```
print("Returns the average value")
print (df.mean())
```

std()
Returns the Bressel standard deviation of the numerical columns.

*print (df.std())*

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

| Sr.No. | Function | Description |
|---|---|---|
| 1 | count() | Number of non-null observations |
| 2 | sum() | Sum of values |
| 3 | mean() | Mean of Values |
| 4 | median() | Median of Values |
| 5 | mode() | Mode of values |
| 6 | std() | Standard Deviation of the Values |
| 7 | min() | Minimum Value |
| 8 | max() | Maximum Value |
| 9 | abs() | Absolute Value |
| 10 | prod() | Product of Values |
| 11 | cumsum() | Cumulative Sum |
| 12 | cumprod() | Cumulative Product |

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

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

# Python Pandas – Re-indexing

Reindexing changes the row labels and column labels of a DataFrame. To reindex means to conform the data to match a given set of labels along a particular axis.

Multiple operations can be accomplished through indexing like −

Reorder the existing data to match a new set of labels.

Insert missing value (NA) markers in label locations where no data for the label existed.

```
import pandas as pd
import numpy as np

N=20

df = pd.DataFrame({
   'A': pd.date_range(start='2016-01-01',periods=N,freq='D'),
   'x': np.linspace(0,stop=N-1,num=N),
   'y': np.random.rand(N),
   'C': np.random.choice(['Low','Medium','High'],N).tolist(),
   'D': np.random.normal(100, 10, size=(N)).tolist()
})

#reindex the DataFrame
df_reindexed = df.reindex(index=[0,2,5], columns=['A', 'C', 'B'])

print( df_reindexed)
```

## Python Pandas - Iteration

The behavior of basic iteration over Pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. Other data structures, like DataFrame and Panel, follow the dict-like convention of iterating over the keys of the objects.

In short, basic iteration (for i in object) produces −

**Series – values**

**DataFrame – column labels**

### Iterating a DataFrame

Iterating a DataFrame gives column names. Let us consider the following example to understand the same.

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

```
N=20
df = pd.DataFrame({
  'A': pd.date_range(start='2016-01-01',periods=N,freq='D'),
  'x': np.linspace(0,stop=N-1,num=N),
  'y': np.random.rand(N),
  'C': np.random.choice(['Low','Medium','High'],N).tolist(),
  'D': np.random.normal(100, 10, size=(N)).tolist()
  })
```

```
for eachCols in df:
   print (eachCols)
```

To iterate over the rows of the DataFrame, we can use the following functions −

iteritems() − to iterate over the (key,value) pairs

iterrows() − iterate over the rows as (index,series) pairs

itertuples() − iterate over the rows as namedtuples

### iteritems()

Iterates over each column as key, value pair with label as key and column value as a Series object.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(4,3),columns=['col1','col2','col3'])
for key,value in df.iteritems():
  print key,value
```

**iterrows**()

iterrows() returns the iterator yielding each index value along with a series containing the data in each row.

```
for row_index,row in df.iterrows():
  print( row_index,row)
```

Note − Because iterrows() iterate over the rows, it doesn't preserve the data type across the row. 0,1,2 are the row indices and col1,col2,col3 are column indices.

**itertuples**()

itertuples() method will return an iterator yielding a named tuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
for row in df.itertuples():
   print row
```

**Note** − Do not try to modify any object while iterating. Iterating is meant for reading and the iterator returns a copy of the original object (a view), thus the changes will not reflect on the original object.

## *Python Pandas - Sorting*

There are two kinds of sorting available in Pandas. They are −

By label
By Actual Value
Let us consider an example with an output.

```
import pandas as pd
import numpy as np

unsorted_df=pd.DataFrame(np.random.randn(10,2), index=[1,4,6,2,3,5,9,8,0,7],
columns=['col2','col1'])
print unsorted_df
```

In unsorted_df, the labels and the values are unsorted. Let us see how these can be sorted.

By Label
Using the sort_index() method, by passing the axis arguments and the order of sorting, DataFrame can be sorted. By default, sorting is done on row labels in ascending order.

```
import pandas as pd
import numpy as np

unsorted_df = pd.DataFrame(np.random.randn(10,2), index=[1,4,6,2,3,5,9,8,0,7], columns =
['col2','col1'])

sorted_df=unsorted_df.sort_index()
print sorted_df
```

**Order of Sorting**

By passing the Boolean value to ascending parameter, the order of the sorting can be controlled. Let us consider the following example to understand the same.

```
import pandas as pd
import numpy as np

unsorted_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],colu
  mns = ['col2','col1'])
```

```
sorted_df = unsorted_df.sort_index(ascending=False)
print sorted_df
```

**Sort the Columns:**

By passing the axis argument with a value 0 or 1, the sorting can be done on the column labels. By default, axis=0, sort by row. Let us consider the following example to understand the same.

```
import pandas as pd
import numpy as np
 unsorted_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],colu
   mns = ['col2','col1'])

sorted_df=unsorted_df.sort_index(axis=1)

print (sorted_df)
```

By Value
Like index sorting, sort_values() is the method for sorting by values. It accepts a 'by' argument which will use the column name of the DataFrame with which the values are to be sorted.

```
import pandas as pd
import numpy as np

unsorted_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})
   sorted_df = unsorted_df.sort_values(by='col1')

print sorted_df
```

Observe, col1 values are sorted and the respective col2 value and row index will alter along with col1. Thus, they look unsorted.

'by' argument takes a list of column values.

## Python Pandas - Working with Text Data

Let us now create a Series and see how all the above functions work.

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])
print (s)
```

Its output is as follows −

```
0          Tom
1   William Rick
2         John
3       Alber@t
4          NaN
5         1234
6    Steve Smith
dtype: object
```

**lower()**

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])

print("Lower Case::")
print (s.str.lower())
```

Its output is as follows –

```
0          tom
1          william rick
2          john
3          alber@t
4          NaN
5          1234
6          steve smith
dtype: object
```

**upper()**

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])

print (s.str.upper())
```

Its output is as follows –

```
0          TOM
1          WILLIAM RICK
2         JOHN
3          ALBER@T
4          NaN
5         1234
6          STEVE SMITH
dtype: object
```

**len()**

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])
print (s.str.len())
```

Its output is as follows −

```
0    3.0
1   12.0
2    4.0
3    7.0
4    NaN
5    4.0
6   10.0
dtype: float64
```

**strip()**

```
import pandas as pd
import numpy as np
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s
print ("After Stripping:")
print (s.str.strip())
```

Its output is as follows −

```
0          Tom
1          William Rick
2         John
3       Alber@t
dtype: object

After Stripping:
0          Tom
1          William Rick
2         John
3          Alber@t
dtype: object
```

**split(pattern)**

```
import pandas as pd
import numpy as np
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s
```

```
print ("Split Pattern:")
print s.str.split(' ')
```

Its output is as follows −

```
0          Tom
1          William Rick
2         John
3       Alber@t
dtype: object

Split Pattern:
0   [Tom, , , , , , , , , , ]
1   [, , , , , William, Rick]
2   [John]
3   [Alber@t]
dtype: object
```

## cat(sep=pattern)

```
import pandas as pd
import numpy as np
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s.str.cat(sep='_')
```

Its output is as follows −

```
Tom _ William Rick_John_Alber@t
```

**contains ()**

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s.str.contains(' ')
```

Its output is as follows −

```
0   True
1   True
2   False
3   False
dtype: bool
```

**replace(a,b)**

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s
print ("After replacing @ with $:")
```

*print s.str.replace('@','$')*

Its output is as follows −

```
0   Tom
1   William Rick
2   John
3   Alber@t
dtype: object

After replacing @ with $:
0   Tom
1   William Rick
2   John
3   Alber$t
dtype: object
```

repeat(value)

*import pandas as pd*
*s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])*
*print s.str.repeat(2)*

Its output is as follows −

```
0          Tom              Tom
1          William Rick     William Rick
2           John            John
3           Alber@t         Alber@t
dtype: object
```

**count(pattern)**

*import pandas as pd*

*s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])*

*print ("The number of 'm's in each string:")*
*print s.str.count('m')*

Its output is as follows −

```
The number of 'm's in each string:
0   1
1   1
2   0
3   0
```

**startswith(pattern)**

*import pandas as pd*

```
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print ("Strings that start with 'T':")
print s.str. startswith ('T')
```

Its output is as follows −

```
0  True
1  False
2  False
3  False
dtype: bool
```

**endswith(pattern)**

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print ("Strings that end with 't':")
print s.str.endswith('t')
```

Its output is as follows −

```
Strings that end with 't':
0  False
1  False
2  False
3  True
dtype: bool
```

**find(pattern)**

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s.str.find('e')
```

Its output is as follows −

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

"-1" indicates that there no such pattern available in the element.

**findall(pattern)**

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s.str.findall('e')
```

Its output is as follows −

```
0 []
1 []
2 []
3 [e]
dtype: object
```

Null list([ ]) indicates that there is no such pattern available in the element.

**swapcase()**

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])
print s.str.swapcase()
```

Its output is as follows −

```
0  tOM
1  wILLIAM rICK
2  jOHN
3  aLBER@T
dtype: object
```

**islower()**

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])
print s.str.islower()
```

Its output is as follows −

```
0  False
1  False
2  False
3  False
dtype: bool
```

**isupper()**

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])
print s.str.isupper()
```

Its output is as follows −

0  False
1  False
2  False
3  False
dtype: bool

**isnumeric()**

```
import pandas as pd
s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])
print s.str.isnumeric()
```

Its output is as follows −

0  False
1  False
2  False
3  False
dtype: bool