

Python Pandas Module

Dr. B.S.R.V. Prasad

Department of Mathematics

School of Advanced Sciences

Vellore Institute of Technology

Vellore - 632014, TN, India



Jupyter Notebooks used for demonstration Pandas class lectures can be downloaded from below URLs:

Pandas Basics: https://bsrvp.github.io/PythonClass/Pandas_Basics.ipynb

Pandas Function Applications:

https://bsrvp.github.io/PythonClass/Python_Function_Applications.ipynb

Pandas Plotting:

https://bsrvp.github.io/PythonClass/Pandas_Plotting.ipynb

Data File: <https://bsrvp.github.io/PythonClass/WQ.xlsx>

Plotting in Python: https://bsrvp.github.io/PythonClass/Plotting_and_Visualization.ipynb

Python 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.

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.

Key Features of Pandas

- 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.

Pandas deals with the following three data structures

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Dimension & Description

- 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.

Dimension & Description

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

Dimension & Description

- Building and handling two or more dimensional arrays is a tedious task, burden is placed on the user to consider the orientation of the data set when writing functions.
- But using Pandas data structures, the mental effort of the user is reduced.
- For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1.

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.

Pandas 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

Pandas DataFrame

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

Name	Age	Gender	Rating		Column	Type
Steve	32	Male	3.45		Name	String
Lia	28	Female	4.6		Age	Integer
Vin	45	Male	3.9		Gender	String
Katie	38	Female	2.78		Rating	Float

Key Points: Heterogeneous data, Size Mutable, Data Mutable

Pandas Panel

- Panel is a three-dimensional data structure with heterogeneous data.
- It is hard to represent the panel in graphical representation.
- But a panel can be illustrated as a container of DataFrame.

Key Points: Heterogeneous data, Size Mutable, Data Mutable

Python Pandas - Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

pandas.Series

A pandas Series can be created using the following constructor –

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

Python Pandas - Series

The parameters of the constructor are as follows –

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

Python Pandas - 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.

Example

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

Its output is as follows –

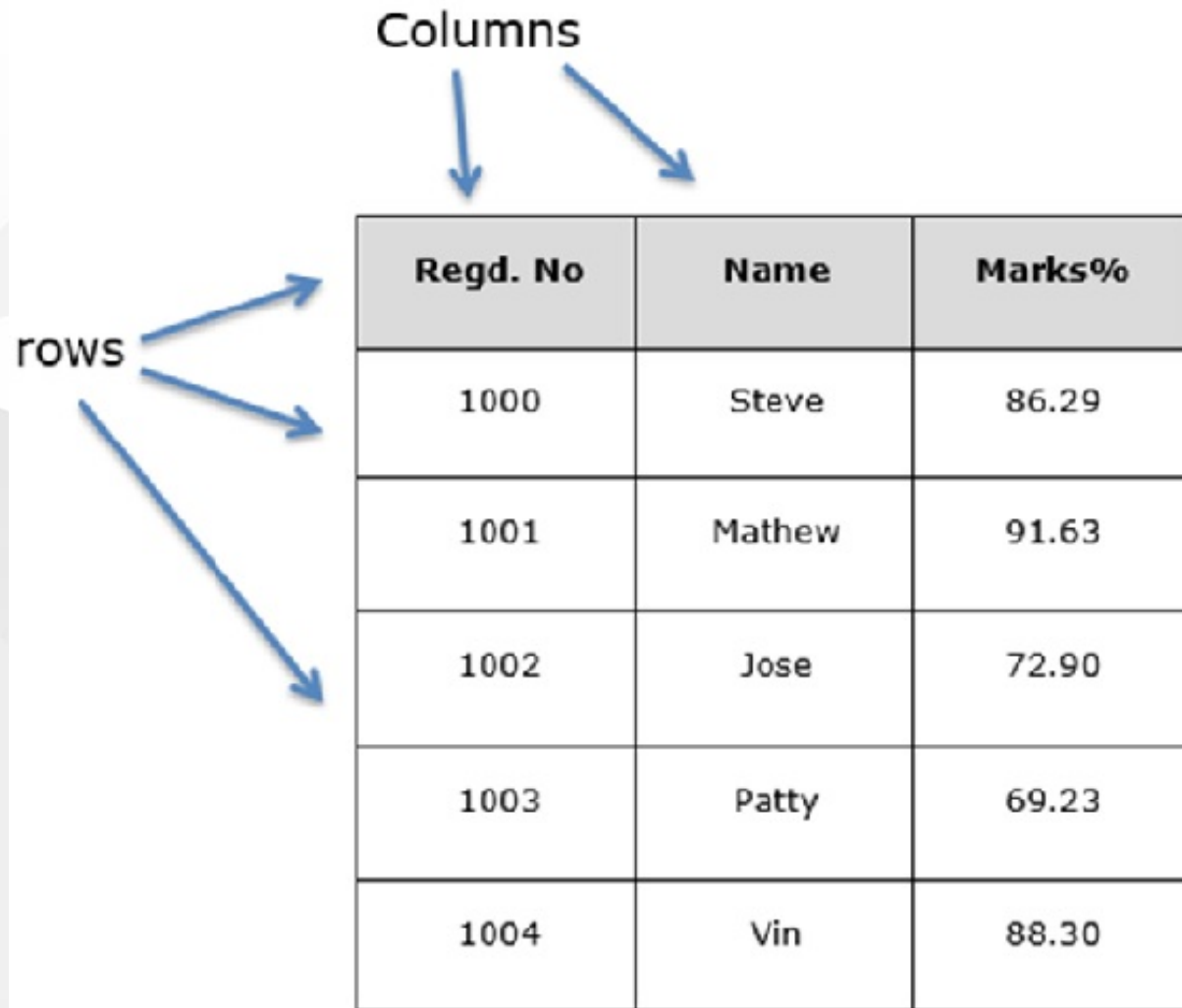
```
Series([], dtype: float64)
```

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
- Labelled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

Structure

Let us assume that we are creating a data frame with student's data.

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)
```

pandas.DataFrame

The parameters of the constructor are as follows –

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

```
#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.

Example 1

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

Its output is as follows –

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


Example 2

```
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

Example 3

```
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.

Example 1

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

Its output is as follows –

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

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

Example 2

Let us now create an indexed DataFrame using arrays.

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

Its **output** is as follows –

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

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(df1)
print(df2)
```

Its **output** is as follows –

```
#df1 output
```

	a	b
first	1	2
second	5	10

```
#df2 output
```

	a	b1
first	1	NaN
second	5	NaN

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

Create a DataFrame from Dict of Series

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

Example

```
import pandas as pd

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

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

Its **output** is as follows –

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

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

```
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.

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)

# 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

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.

```
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.loc['b'])
```

Its **output** is as follows –

```
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.

```
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.

Deletion of Columns

Use index label to delete or drop columns from a DataFrame using `pd.drop(column_name, axis = 1)`. If label is duplicated, then multiple columns will be 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('b',axis=1)

print(df)
```


Its **output** is as follows –

	a
0	1
1	3
0	5
1	7

Python Pandas - Function Application

To apply your own or another library's functions to Pandas objects, you should be aware of the three important methods. The methods have been discussed below. The appropriate method to use depends on whether your function expects to operate on an entire DataFrame, row- or column-wise, or element wise.

- Table wise Function Application: `pipe()`
- Row or Column Wise Function Application: `apply()`
- Element wise Function Application: `applymap()`

Table-wise Function Application

Custom operations can be performed by passing the function and the appropriate number of parameters as pipe arguments. Thus, operation is performed on the whole DataFrame.

For example, add a value 2 to all the elements in the DataFrame. Then,

adder function

The adder function adds two numeric values as parameters and returns the sum.

```
def adder(ele1,ele2):  
    return ele1+ele2
```

We will now use the custom function to conduct operation on the DataFrame.

```
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])  
df.pipe(adder,2)
```

The full program –

```
import pandas as pd  
import numpy as np  
  
def adder(ele1,ele2):  
    return ele1+ele2  
  
df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])  
print(df.pipe(adder,2))
```

Its **output** is as follows –

	col1	col2	col3
0	2.176704	2.219691	1.509360
1	2.222378	2.422167	3.953921
2	2.241096	1.135424	2.696432
3	2.355763	0.376672	1.182570
4	2.308743	2.714767	2.130288

Row or Column Wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the **apply()** method, which, like the descriptive statistics methods, takes an optional axis argument. By default, the operation performs column wise, taking each column as an array-like.

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), columns=['col1', 'col2', 'col3'])
print(df.apply(np.mean))
```

Its **output** is as follows –

```
col1    0.778735  
col2   -0.507344  
col3   -0.963479  
dtype: float64
```

By passing **axis** parameter, operations can be performed row wise.

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
print(df.apply(np.mean,axis=1))
```

Its output is as follows –

```
0    0.308779
1   -0.059966
2    1.090742
3    0.563616
4    0.616710
dtype: float64
```


Example 3

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), columns=['col1', 'col2', 'col3'])
df1 = df.apply(lambda x: x.max() - x.min())
print(df1)
```

Its **output** is as follows –

```
col1    2.899872
col2    2.941797
col3    2.500316
dtype: float64
```

Example 4

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3),columns=['col1','col2','col3'])
df1 = df.apply(lambda x: x.max() - x.min(), axis=1)
print(df1)
```

Its output is as follows –

```
0    1.435421
1    1.685351
2    0.552591
3    0.338889
4    2.111921
dtype: float64
```

Element Wise Function Application

Not all functions can be vectorized (neither the NumPy arrays which return another array nor any value), the methods `applymap()` on DataFrame (and analogously `map()` on Series) accepts any Python function taking a single value and returning a single value.

Example 1

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5,3), columns=['col1', 'col2', 'col3'])

# My custom function
df1 = df['col1'].map(lambda x:x*100)
print(df1)
```

Its **output** is as follows –

```
0    -28.092383  
1     15.497209  
2     48.183373  
3    -11.853294  
4    -71.943119
```

```
Name: col1, dtype: float64
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), columns=['col1', 'col2', 'col3'])
df1 = df.applymap(lambda x:x*100)
print(df1)
```

Its output is as follows –

	col1	col2	col3
0	-2.272304	200.601032	1.549926
1	68.475810	125.236636	234.753324
2	78.652754	68.507668	-92.210604
3	112.611038	2.505065	-168.766786
4	192.413533	-29.642830	146.395853

Python Pandas - Reindexing

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.

Example

```
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)
```

Its **output** is as follows –

	A	C	B
0	2016-01-01	Low	NaN
2	2016-01-03	High	NaN
5	2016-01-06	Low	NaN

Reindex to Align with Other Objects

You may wish to take an object and reindex its axes to be labeled the same as another object. Consider the following example to understand the same.

Example

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(10,3), columns=['col1', 'col2', 'col3'])
df2 = pd.DataFrame(np.random.randn(7,3), columns=['col1', 'col2', 'col3'])

df1 = df1.reindex_like(df2)
print(df1)
```

Its **output** is as follows –

	col1	col2	col3
0	-2.467652	-1.211687	-0.391761
1	-0.287396	0.522350	0.562512
2	-0.255409	-0.483250	1.866258
3	-1.150467	-0.646493	-0.222462
4	0.152768	-2.056643	1.877233
5	-1.155997	1.528719	-1.343719
6	-1.015606	-1.245936	-0.295275

Note – Here, the **df1** DataFrame is altered and reindexed like **df2**. The column names should be matched or else NAN will be added for the entire column label.

Filling while ReIndexing

`reindex()` takes an optional parameter `method` which is a filling method with values as follows –

- `pad/ffill` – Fill values forward
- `bfill/backfill` – Fill values backward
- `nearest` – Fill from the nearest index values

Example

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(6,3),columns=['col1','col2','col3'])
df2 = pd.DataFrame(np.random.randn(2,3),columns=['col1','col2','col3'])

# Padding NAN's
print(df2.reindex_like(df1))

# Now Fill the NAN's with preceding Values
print ("Data Frame with Forward Fill:")
print(df2.reindex_like(df1,method='ffill'))
```

Its output is as follows –

	col1	col2	col3
0	1.311620	-0.707176	0.599863
1	-0.423455	-0.700265	1.133371
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

Data Frame **with** Forward Fill:

	col1	col2	col3
0	1.311620	-0.707176	0.599863
1	-0.423455	-0.700265	1.133371
2	-0.423455	-0.700265	1.133371
3	-0.423455	-0.700265	1.133371
4	-0.423455	-0.700265	1.133371
5	-0.423455	-0.700265	1.133371

Note – The last four rows are padded.

Limits on Filling while Reindexing

The limit argument provides additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches. Let us consider the following example to understand the same –

Example

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(6,3), columns=['col1', 'col2', 'col3'])
df2 = pd.DataFrame(np.random.randn(2,3), columns=['col1', 'col2', 'col3'])

# Padding NAN's
print(df2.reindex_like(df1))

# Now Fill the NAN's with preceding Values
print ("Data Frame with Forward Fill limiting to 1:")
print(df2.reindex_like(df1, method='ffill', limit=1))
```

Its **output** is as follows –

	col1	col2	col3
0	0.247784	2.128727	0.702576
1	-0.055713	-0.021732	-0.174577
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

Data Frame **with** Forward Fill limiting to **1**:

	col1	col2	col3
0	0.247784	2.128727	0.702576
1	-0.055713	-0.021732	-0.174577
2	-0.055713	-0.021732	-0.174577
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

Note – Observe, only the 7th row is filled by the preceding 6th row. Then, the rows are left as they are.

Renaming

The `rename()` method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

Let us consider the following example to understand this –

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(6,3), columns=['col1', 'col2', 'col3'])
print df1

print ("After renaming the rows and columns:")
print(df1.rename(columns={'col1' : 'c1', 'col2' : 'c2'},)
index = {0 : 'apple', 1 : 'banana', 2 : 'durian'})
```


Its **output** is as follows –

	col1	col2	col3
0	0.486791	0.105759	1.540122
1	-0.990237	1.007885	-0.217896
2	-0.483855	-1.645027	-1.194113
3	-0.122316	0.566277	-0.366028
4	-0.231524	-0.721172	-0.112007
5	0.438810	0.000225	0.435479

After renaming the rows **and** columns:

	c1	c2	col3
apple	0.486791	0.105759	1.540122
banana	-0.990237	1.007885	-0.217896
durian	-0.483855	-1.645027	-1.194113
3	-0.122316	0.566277	-0.366028
4	-0.231524	-0.721172	-0.112007
5	0.438810	0.000225	0.435479

The `rename()` method provides an **inplace** named parameter, which by default is False and copies the underlying data. Pass **inplace=True** to rename the data in place.

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
- **Panel** – item 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 col in df:
    print(col)
```

Its **output** is as follows –

```
A  
C  
D  
x  
y
```

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

- **items()** – to iterate over the (key,value) pairs
- **iterrows()** – iterate over the rows as (index,series) pairs
- **itertuples()** – iterate over the rows as namedtuples

items()

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.items():
    print(key,value)
```

Its **output** is as follows –

```
col1 0    0.802390
```

```
1    0.324060
```

```
2    0.256811
```

```
3    0.839186
```

```
Name: col1, dtype: float64
```

```
col2 0    1.624313
```

```
1   -1.033582
```

```
2    1.796663
```

```
3    1.856277
```

```
Name: col2, dtype: float64
```

```
col3 0   -0.022142
```

```
1   -0.230820
```

```
2    1.160691
```

```
3   -0.830279
```

```
Name: col3, dtype: float64
```

Observe, each column is iterated separately as a key-value pair in a Series.

iterrows()

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

```
import pandas as pd
import numpy as np

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

Its **output** is as follows –

```
0  col1    1.529759
   col2    0.762811
   col3   -0.634691
Name: 0, dtype: float64

1  col1   -0.944087
   col2    1.420919
   col3   -0.507895
Name: 1, dtype: float64

2  col1   -0.077287
   col2   -0.858556
   col3   -0.663385
Name: 2, dtype: float64

3  col1   -1.638578
   col2    0.059866
   col3    0.493482
Name: 3, dtype: float64
```

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.

```
import pandas as pd
import numpy as np

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

Its **output** is as follows –

```
Pandas(Index=0, col1=1.5297586201375899, col2=0.76281127433814944, col3=-0.6346908238310438)
```

```
Pandas(Index=1, col1=-0.94408735763808649, col2=1.4209186418359423, col3=-0.50789517967096232)
```

```
Pandas(Index=2, col1=-0.07728664756791935, col2=-0.85855574139699076, col3=-0.6633852507207626)
```

```
Pandas(Index=3, col1=0.65734942534106289, col2=-0.95057710432604969, col3=0.80344487462316527)
```

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.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(4,3), columns = ['col1', 'col2', 'col3'])

for index, row in df.iterrows():
    row['a'] = 10
print(df)
```

Its **output** is as follows –

	col1	col2	col3
0	-1.739815	0.735595	-0.295589
1	0.635485	0.106803	1.527922
2	-0.939064	0.547095	0.038585
3	-1.016509	-0.116580	-0.523158

Observe, no changes reflected.