Review of Python Pandas
Based on CBSE Curriculum
Informatics Practices Class-12

CHAPTER-1

By:
Neha Tyagi, PGT CS
KV no-5 2nd Shift, Jaipur
Jaipur Region
Python Pandas (A Review)

• Data Processing is the most important part of Data Analysis. Because data is not available every time in desired format.

• Before analyzing the data it needs various types of processing like - Cleaning, Restructuring or merging etc.

• There are many tools available in python to process the data fast Like-Numpy, Scipy, Cython and Pandas.

• Pandas are built on the top of Numpy.

• In this chapter we will learn about the basic concepts of Python Pandas Data Series and DataFrames which we learnt in class -11.
Python Pandas

- Pandas is an open-source library of python providing high-performance data manipulation and analysis tool using its powerful data structure.
- Pandas provides rich set of functions to process various types of data.
- During data analysis it is very important to make it confirm that you are using correct data types otherwise you may face some unexpected errors.
- Some of the pandas supporting data types are as follows -

<table>
<thead>
<tr>
<th>Pandas dtype</th>
<th>Python type</th>
<th>NumPy type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>str</td>
<td>string_, unicode_</td>
<td>Text</td>
</tr>
<tr>
<td>int64</td>
<td>int</td>
<td>int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64</td>
<td>Integer numbers</td>
</tr>
<tr>
<td>float64</td>
<td>float</td>
<td>float_, float16, float32, float64</td>
<td>Floating point numbers</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>bool_</td>
<td>True/False values</td>
</tr>
<tr>
<td>datetime64</td>
<td>NA</td>
<td>datetime64[ns]</td>
<td>Date and time values</td>
</tr>
<tr>
<td>timedelta[ns]</td>
<td>NA</td>
<td>NA</td>
<td>Differences between two datetimes</td>
</tr>
<tr>
<td>category</td>
<td>NA</td>
<td>NA</td>
<td>Finite list of text values</td>
</tr>
</tbody>
</table>
Pandas Series

- **Series** is the primary building block of Pandas.
- **Series** is a labeled *One-Dimensional Array* which can hold any type of data.
- Data of Series is always *mutable*. It means, it can be changed.
- But the size of data of Series is size *immutable*, means can not be changed.
- It can be seen as a data structure with two arrays: one functioning as the *index* (Labels) and the other one contains the actual data.
- In Series, row labels are also called the *index*.
- Let's take some data which can be considered as series -

```python
Num = [23, 54, 34, 44, 35, 66, 27, 88, 69, 54]  # a list with homogeneous data
Emp = ['A V Raman', 35, 'Finance', 45670.00]  # a list with heterogeneous data
Marks = {'ELENA JOSE': 450, 'PARAS GUPTA': 467, 'JOEFFIN JOSEPH': 480}  # a dictionary
Num1 = (23, 54, 34, 44, 35, 66, 27, 88, 69, 54)  # a tuple with homogeneous data
Std = ('AKYHA KUMAR', 78.0, 79.0, 89.0, 88.0, 91.0)  # a list with heterogeneous data
```
Creation of Series Objects

There are many ways to create series type object.

1. Using Series ( )-

```
<Series Object> = pandas.Series() it will create empty series.
```

```
>>> import pandas as pd
>>> ob = pd.Series()
>>> ob
Series([], dtype: float64)
```

2. Non-empty series creation–

Import pandas as pd

```
<Series Object> = pd.Series(data, index=idx) where data can be python sequence, ndarray, python dictionary or scaler value.
```

```
>>> import pandas as pd
>>> ob = pd.Series(range(5))
>>> ob
0    0
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> import pandas as pd
>>> obj=pd.Series([3,5,4,4.5])
>>> obj
0    3.0
1    5.0
2    4.0
3    4.5
dtype: float64
```
Series Objects creation

1. Creation of series with Dictionary-

```python
>>> import pandas as pd
>>> obj=pd.Series({'Jan':31,'Feb':28,'Mar':31})
>>> obj
Jan   31
Feb   28
Mar   31
dtype: int64
```

2. Creation of series with Scalar value-

```python
>>> import pandas as pd
>>> a=pd.Series(10,index=range(0,3))
>>> a
0  10
1  10
2  10
dtype: int64

>>> import pandas as pd
>>> b=pd.Series(15,index=range(1,6,2))
>>> b
1  15
3  15
5  15
dtype: int64

>>> import pandas as pd
>>> c=pd.Series('Welcome to BBK', index=['Hema','Rahul','Anup'])
>>> c
Hema    Welcome to BBK
Rahul    Welcome to BBK
Anup    Welcome to BBK
dtype: object
```
Creation of Series Objects – Additional functionality

1. When it is needed to create a series with missing values, this can be achieved by filling missing data with a NaN (“Not a Number”) value.

```python
>>> import pandas as pd
>>> import numpy as np
>>> ob=pd.Series([6.5, np.NaN, 2.34])
>>> ob
0   6.50
1   NaN
2   2.34
dtype: float64
```

2. Index can also be given as-

```python
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3), index=[x for x in 'abcde'])
>>> s
a   1
b   4
c   7
d  10
e  13
dtype: int64

Loop is used to give Index
```
Creation of Series Objects – Additional functionality

3. Dtype can also be passed with Data and index

```python
>>> import pandas as pd
>>> import numpy as np
>>> ob=pd.Series(data=arr,index=mon,dtype=np.float64)
>>> ob
Jan    31.0
Feb    28.0
Mar    31.0
Apr    30.0
dtype: float64
```

**Important**: it is not necessary to have unique indices but it will give error when search will be according to index.

4. Mathematical function/Expression can also be used-

```python
>>> import pandas as pd
>>> import numpy as np
>>> a=np.arange(9,13)
>>> a
array([ 9, 10, 11, 12])
>>> ob=pd.Series(index=a, data=a**2)
>>> ob
9    81
10   100
11  121
12  144
dtype: int32
```
3. Some common attributes-

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.index</td>
<td>Returns index of the series</td>
</tr>
<tr>
<td>Series.values</td>
<td>Returns ndarray</td>
</tr>
<tr>
<td>Series.dtype</td>
<td>Returns dtype object of the underlying data</td>
</tr>
<tr>
<td>Series.shape</td>
<td>Returns tuple of the shape of underlying data</td>
</tr>
<tr>
<td>Series.nbytes</td>
<td>Return number of bytes of underlying data</td>
</tr>
<tr>
<td>Series.ndim</td>
<td>Returns the number of dimension</td>
</tr>
<tr>
<td>Series.size</td>
<td>Returns number of elements</td>
</tr>
<tr>
<td>Series.intensize</td>
<td>Returns the size of the dtype</td>
</tr>
<tr>
<td>Series.hasnans</td>
<td>Returns true if there are any NaN</td>
</tr>
<tr>
<td>Series.empty</td>
<td>Returns true if series object is empty</td>
</tr>
</tbody>
</table>
Series Object Attributes

```python
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3), index=[x for x in 'abcde'])
>>> s.index
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> s.values
array([ 1,  4,  7, 10, 13], dtype=int64)
>>> s.shape
(5,)
>>> s.size
5
>>> s.nbytes
40
>>> s.ndim
1
>>> s.itemsize
```
Accessing Series Object

For Object slicing, follow the following syntax-

```
<objectName>[<start>:<stop>:<step >]
```
Operations on Series Object

1. Elements modification:

\[ \text{<series object>[index]} = \text{<new_data_value>} \]

```python
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3), index=[x for x in 'abcde'])
```

```python
>>> s
a   1
b   4
c   7
d  10
e  13
dtype: int64
```

```python
>>> s['c']=17
```

```python
>>> s
a   1
b   4
c  17
d  10
e  13
dtype: int64
```

```python
>>> s[2:4]=100
```

```python
>>> s
a   1
b   4
c  100
d  100
e  13
dtype: int64
```

```python
>>> s[1:5:2]=100
```

```python
>>> s
a   1
b  100
c   7
d  100
e  13
dtype: int64
```

To change individual value

To change value in a certain slice
Operations on Series Object

1. It is possible to change indexes

\[ \text{<series object>.<index]} = \text{<new_index_array>} \]

```python
>>> import pandas as pd
>>> s = pd.Series(range(1,15,3), index=[x for x in 'abcde'])
```

```python
>>> s
a    1
b    4
c    7
d   10
e   13
dtype: int64
```

```python
>>> s.index=['u','v','w','x','y']
```

```python
>>> s
u    1
v    4
w    7
x   10
y   13
dtype: int64
```

Here, indexes got changed.

Neha Tyagi, KV5 Jaipur II shift
head() and tail() Function

1. head(<n>) function fetch first n rows from a pandas object. If you do not provide any value for n, will return first 5 rows.

2. tail(<n>) function fetch last n rows from a pandas object. If you do not provide any value for n, will return last 5 rows.

```python
>>> import pandas as pd
>>> import math
>>> s=pd.Series(data=[math.sqrt(x) for x in range(1,10)],index=[x for x in range(1,10)])

>>> s
0  1.000000
1  1.414214
2  1.732051
3  2.000000
4  2.236068
5  2.449490
6  2.645751
7  2.828427
8  3.000000
9  3.000000
dtype: float64

>>> s.head(6)
0  1.000000
1  1.414214
2  1.732051
3  2.000000
4  2.236068
5  2.449490
6  2.449490
dtype: float64

>>> s.tail(7)
3  1.732051
4  2.000000
5  2.236068
6  2.449490
7  2.645751
8  2.828427
9  3.000000
dtype: float64

>>> s.head()
0  1.000000
1  1.414214
2  1.732051
3  2.000000
4  2.236068
dtype: float64

>>> s.tail()
5  2.236068
6  2.449490
7  2.645751
8  2.828427
9  3.000000
dtype: float64
```
Series Objects - Vector Operations

All these are vector operations

```
>>> s
1    11
2     2
3     3
4     4
dtype: int64
```

```
>>> s+2
1    13
2    14
3    15
4    16
dtype: int64
```

```
>>> s*3
1    33
2    36
3    39
4    42
dtype: int64
```

```
>>> s**2
1    121
2    144
3    169
4    196
dtype: int64
```

```
>>> s>13
1   False
2   False
3   False
4    True
dtype: bool
```

Series Objects - Arithmetic Operations

Arithmetic operation is possible on objects of same index otherwise will result as NaN.

```
>>> s
1    11
2     2
3     3
4     4
dtype: int64
```

```
>>> s1
1     1
2     2
3     3
4     4
dtype: int64
```

```
>>> s+s1
1     32
2     34
3     36
4     38
dtype: int64
```

```
>>> s*s1
1    231
2    264
3    299
4    336
dtype: int64
```

```
>>> s/s1
1  0.523810
2  0.545455
3  0.565217
4  0.583333
dtype: float64
```

We can also store these results in other objects.
Entries Filtering

<seriesObject> <series - boolean expression >

```
>>> s
1   1.000000
2   1.414214
3   1.732051
4   2.000000
dtype: float64
>>> s<2
1   True
2   True
3   True
4   False
dtype: bool
>>> s[s<2]
1   1.000000
2   1.414214
3   1.732051
dtype: float64
>>> s[s>=2]
4   2.0
   dtype: float64
```
Difference between NumPy array  Series objects

1. In case of ndarray, vector operation is possible only when ndarray are of similar shape. Whereas in case of series object, it will be aligned only with matching index otherwise NaN will be returned.

```python
>>> import numpy as np
>>> a=np.array([1,2,3])
>>> b=np.array([1,2,3,45,5])
>>> a+b
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in <module>
    a+b
ValueError: operands could not be broadcast together with shapes (3,) (5,)
```

2. In ndarray, index always starts from 0 and always numeric. Whereas, in series, index can be of any type including number and not necessary to start from 0.
DataFrame

- Pandas का मुख्य object **DataFrame** होता है और यह pandas का सबसे अधिक प्रयोग किया जाने वाला Data Structure है।
- **DataFrame** एक *Two-Dimensional Array* होता है जो किसी भी data type को hold कर सकती है और यह tabular format में data को store करता है।
- Finance, Statistics, Social Science और कई engineering branch में इसका प्रयोग अधिकता में किया जाता है।
- DataFrame में data और इसका size दोनों ही mutable होते हैं अर्थात इन्हें बदला जा सकता है।
- DataFrame में दो विभिन्न indexes होते हैं - *row index* और *column index*।

**A DataFrame with two-dimensional array with heterogeneous data.**

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
<th>BirthRate</th>
<th>UpdateDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1,379,750,000</td>
<td>14.00</td>
<td>2016-08-11</td>
</tr>
<tr>
<td>India</td>
<td>1,330,780,000</td>
<td>21.76</td>
<td>2016-08-11</td>
</tr>
<tr>
<td>United States</td>
<td>324,882,000</td>
<td>13.82</td>
<td>2016-08-11</td>
</tr>
<tr>
<td>Indonesia</td>
<td>260,581,000</td>
<td>18.84</td>
<td>2016-01-07</td>
</tr>
<tr>
<td>Brazil</td>
<td>206,918,000</td>
<td>18.43</td>
<td>2016-08-11</td>
</tr>
<tr>
<td>Pakistan</td>
<td>194,754,000</td>
<td>27.62</td>
<td>2016-08-11</td>
</tr>
</tbody>
</table>
Creation and presentation of DataFrame

- DataFrame object can be created by passing a data in 2D format.

```python
import pandas as pd
<dataFrameObject> = pd.DataFrame(<a 2D Data Structure>, [columns=<column sequence>], [index=<index sequence>])
```

- You can create a DataFrame by various methods by passing data values. Like-
  - 2D dictionaries
    - 2D ndarrays
    - Series type object
    - Another DataFrame object
Creation of DataFrame from 2D Dictionary

A. Creation of DataFrame from dictionary of List or ndarrays.

```python
>>> import pandas as pd
>>> dict={'Students':['Pratibha','Ritika','Saumya','Aryan','Keshwam','Priyanka'],'Marks':[69,65,64,59,59,40], 'Sports':['TQ','TT','KB','VB','CR','KO']}
>>> df=pd.DataFrame(dict)
>>> df
   Students  Marks  Sports
0    Pratibha   69      TQ
1      Ritika   65      TT
2     Saumya    64      KB
3       Aryan    59      VB
4    Keshwam    59      CR
5   Priyanka    40      KO
```

Indexes are automatically generated by using np.range(n).

Column name are generated from keys of 2D Dictionary.

In the above example, index are automatically generated from 0 to 5 and column name are same as keys in dictionary.
Here, indexes are specified by you.

Meaning, if you specify the sequence of index then index will be the set specified by you only otherwise it will be automatically generated from 0 to n-1.
Creation of DataFrame from 2D Dictionary

B. Creation of DataFrame from dictionary of Dictionaries-

```python
>>> yr2015={'Qtr1':40000, 'Qtr2':35000, 'Qtr3':47000, 'Qtr4':45000}
>>> yr2016={'Qtr1':42000, 'Qtr2':37000, 'Qtr3':49000, 'Qtr4':47000}
>>> yr2017={'Qtr1':43000, 'Qtr2':38000, 'Qtr3':50000, 'Qtr4':48000}
>>> dtFee=pd.DataFrame(kvfee)
>>> dtFee
   2015  2016  2017
Qtr1  40000  42000  43000
Qtr2  35000  37000  38000
Qtr3  47000  49000  50000
Qtr4  45000  47000  48000
```

It is a 2D Dictionary made up of above given dictionaries.

Here, you can get an idea of how index and column name have assigned.

If keys of yr2015, yr2016 and yr2017 were different here then rows and columns of dataframe would have increased and non-matching rows and column would store NaN.
Creation of Dataframe from 2D ndarray

```python
>>> import pandas as pd
>>> import numpy as np
>>> narr=np.array([[1,2,3],[4,5,6]],np.int32)
>>> narr.shape
(2, 3)
>>> dtf=pd.DataFrame(narr)
>>> dtf
     0  1  2
  0  1  2  3
  1  4  5  6
```
column name and index have automatically been generated here.

```python
>>> import pandas as pd
>>> import numpy as np
>>> narr=np.array([[1,2,3],[4,5,6]],np.int32)
>>> dtf=pd.DataFrame(narr, columns=['One','Two','Three'])
>>> dtf
   One  Two  Three
  0  1   2    3
  1  4   5    6
```
Here, user has given column name.

```python
>>> dtf=pd.DataFrame(narr, columns=['One','Two','Three'],index=['A','B'])
>>> dtf
   One  Two  Three
  A  1   2    3
  B  4   5    6
```
Here, column name and index both have been given by user.

Neha Tyagi, KV5 Jaipur, II Shift
Creation of DataFarmer from 2D Dictionary of same Series Object

```python
>>> import pandas as pd
>>> population=pd.Series([35,39,34,64],index=['Class12','Class11','Class10','Class9'])
>>> AvgMarks=pd.Series([350,390,340,400],index=['Class12','Class11','Class10','Class9'])
>>> dict={0:population,1:AvgMarks}
>>> df=pd.DataFrame(dict)
>>> df
   0         1
Class12  35      350
Class11  39      390
Class10  34      340
Class9   64      400
```

It is a 2D Dictionary made up of series given above.

```python
>>> dict={'Population':population,'AverageMarks':AvgMarks}
>>> df=pd.DataFrame(dict)
>>> df
         Population  AverageMarks
Class12   35          350
Class11   39          390
Class10   34          340
Class9    64          400
```

DataFrame object created.

DataFrame object can also be created like this.

Neha Tyagi, KV5 Jaipur, II Shift
Creation of DataFrame from object of other DataFrame

```python
>>> import pandas as pd
>>> import numpy as np
>>> narr=np.array([[1,2,3],[4,5,6]])
>>> dtf=pd.DataFrame(narr,columns=['first','Second','Third'],index=['A','B'])
>>> dtf
   first  Second  Third
A    1      2       3
B    4      5       6

>>> dtf2=pd.DataFrame(dtf)
>>> dtf2
   first  Second  Third
A    1      2       3
B    4      5       6
```

Displaying DataFrame Object

```python
>>> dtf
   first  Second  Third
A    1      2       3
B    4      5       6

>>> dtf2=pd.DataFrame(dtf)
>>> dtf2
   first  Second  Third
A    1      2       3
B    4      5       6
```
DataFrame Attributes

- When we create an object of a DataFrame then all information related to it like size, datatype etc can be accessed by attributes.

  `<DataFrame Object>..<attribute name>`

- Some attributes are -

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>It shows index of dataframe.</td>
</tr>
<tr>
<td>columns</td>
<td>It shows column labels of DataFrame.</td>
</tr>
<tr>
<td>axes</td>
<td>It return both the axes i.e. index and column.</td>
</tr>
<tr>
<td>dtypes</td>
<td>It returns data type of data contained by dataframe.</td>
</tr>
<tr>
<td>size</td>
<td>It returns number of elements in an object.</td>
</tr>
<tr>
<td>shape</td>
<td>It returns tuple of dimension of dataframe.</td>
</tr>
<tr>
<td>values</td>
<td>It return numpy form of dataframe.</td>
</tr>
<tr>
<td>empty</td>
<td>It is an indicator to check whether dataframe is empty or not.</td>
</tr>
<tr>
<td>ndim</td>
<td>Return an int representing the number of axes / array dimensions.</td>
</tr>
<tr>
<td>T</td>
<td>It Transpose index and columns.</td>
</tr>
</tbody>
</table>
DataFrame Attributes

```python
>>> dtf.index
Index(['A', 'B'], dtype='object')

>>> dtf.columns
Index(['first', 'Second', 'Third'], dtype='object')

>>> dtf.axes
[Index(['A', 'B'], dtype='object'), Index(['first', 'Second', 'Third'], dtype='object')]

>>> dtf.dtypes
first    int32
Second   int32
Third    int32
dtype: object

>>> dtf.empty
False

>>> dtf.count()
first     2
Second    2
Third     2
dtype: int64

>>> dtf.size
6

>>> dtf.shape
(2, 3)

>>> dtf.ndim
2

>>> dtf.T
    A  B
first 1  4
Second 2  5
Third  3  6
```

Neha Tyagi, KV5 Jaipur, II Shift
Selecting and Accessing from DataFrame

- Selecting a Column-
  \[
  <\text{DataFrame Object}>[\text{<column name>}] \\
  \text{or} \quad <\text{DataFrame Object}>.\text{<column name>}
  \]

  \[
  <\text{DataFrame Object}>[\text{List of column name }]
  \]

  ```
  >>> df['first']
  <bound method DataFrame.first of first Second Third
  A  1  2  3
  B  4  5  6>
  >>> df['Second']
  A  2
  B  5
  Name: Second, dtype: int32
  ```

  ```
  >>> df[['Second', 'first']]
  Second  first
  A        2  1
  B        5  4
  ```

  We can change the order in column.
## Selection of subset from DataFrame

DataFrameObject.loc[<StartRow> : <EndRow>, <StartCol> : <EndCol>]

```python
>>> df

<table>
<thead>
<tr>
<th></th>
<th>Population</th>
<th>Avg Income</th>
<th>Per Capita Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delhi</td>
<td>1001</td>
<td>45000</td>
<td>44.955045</td>
</tr>
<tr>
<td>Mumbai</td>
<td>2005</td>
<td>56000</td>
<td>27.930175</td>
</tr>
<tr>
<td>Chennai</td>
<td>30236</td>
<td>57000</td>
<td>1.885170</td>
</tr>
<tr>
<td>Kolkata</td>
<td>4662</td>
<td>46000</td>
<td>9.867010</td>
</tr>
</tbody>
</table>

>>> df.loc[\'Delhi\',\:]

<table>
<thead>
<tr>
<th></th>
<th>Population</th>
<th>Avg Income</th>
<th>Per Capita Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delhi</td>
<td>1001.000000</td>
<td>45000.00000</td>
<td>44.955045</td>
</tr>
</tbody>
</table>

>>> df.loc[:\'Population\','Per Capita Income\']

<table>
<thead>
<tr>
<th></th>
<th>Population</th>
<th>Avg Income</th>
<th>Per Capita Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delhi</td>
<td>1001</td>
<td>45000</td>
<td>44.955045</td>
</tr>
<tr>
<td>Mumbai</td>
<td>2005</td>
<td>56000</td>
<td>27.930175</td>
</tr>
<tr>
<td>Chennai</td>
<td>30236</td>
<td>57000</td>
<td>1.885170</td>
</tr>
<tr>
<td>Kolkata</td>
<td>4662</td>
<td>46000</td>
<td>9.867010</td>
</tr>
</tbody>
</table>

>>> df.loc[\'Mumbai\':'Kolkata\',\:]

<table>
<thead>
<tr>
<th></th>
<th>Population</th>
<th>Avg Income</th>
<th>Per Capita Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mumbai</td>
<td>2005</td>
<td>56000</td>
<td>27.930175</td>
</tr>
<tr>
<td>Chennai</td>
<td>30236</td>
<td>57000</td>
<td>1.885170</td>
</tr>
<tr>
<td>Kolkata</td>
<td>4662</td>
<td>46000</td>
<td>9.867010</td>
</tr>
</tbody>
</table>

>>> df.loc[\'Delhi\':'Mumbai\','Population\':'Avg Income\']

<table>
<thead>
<tr>
<th></th>
<th>Population</th>
<th>Avg Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delhi</td>
<td>1001</td>
<td>45000</td>
</tr>
<tr>
<td>Mumbai</td>
<td>2005</td>
<td>56000</td>
</tr>
</tbody>
</table>
```
Selection of subset from DataFrame

```python
>>> dtf.iloc[0:2, 1:3]
     Avg Income  Per Capita Income
Delhi      45000         44.955045
Mumbai     56000         27.930175
```

Selection of an Individual Value from DataFrame

```python
>>> dtf.iloc[0:2, 1:2]
     Avg Income
Delhi     45000
Mumbai     56000
```

```python
>>> dtf
     One  Two  Three
A     1   2    3
B     4   5    6
>>> dtf.Two['B']
5
>>> dtf.iloc['A', 'Three']
3
>>> dtf.Two[0]
2
```
Accessing and modifying values in DataFrame

a) Syntax to add or change a column-

```python
>>> df
   One  Two  Three
A   1    2    3
B   4    5    6
C   7    8    9

>>> df['Four']=44
>>> df
   One  Two  Three  Four
A   1    2    3   44
B   4    5    6   44
C   7    8    9   44
```

A new column will be created because there is no column with the name ‘Four’.

```python
>>> df['Four']=66
>>> df
   One  Two  Three  Four
A   1    2    3   66
B   4    5    6   66
C   7    8    9   66
```

The values of column will get change because there is a column with the name ‘Four’.
Accessing and modifying values in DataFrame

b) Syntax to add or change a row-

\[ <\text{DFObject}> \text{at}[<\text{RowName}>, : ] = <\text{new value}> \]

या

\[ <\text{DFObject}> \text{loc}[<\text{RowName}>, : ] = <\text{new value}> \]

```
>>> dtf.at['D', :]=88
>>> dtf
   One  Two  Three  Four
A    1.0  2.0  3.0  66.0
B    4.0  5.0  6.0  66.0
C    7.0  8.0  9.0  66.0
D   88.0 88.0 88.0 88.0

>>> dtf.at['D', :]=99
>>> dtf
   One  Two  Three  Four
A    1.0  2.0  3.0  66.0
B    4.0  5.0  6.0  66.0
C    7.0  8.0  9.0  66.0
D   99.0 99.0 99.0 99.0
```

A new row will be created because there is no row with the name ‘D’.

The values of row will get change because there is a row with the name ‘D’.

Neha Tyagi, KV5 Jaipur, II Shift
Accessing and modifying values in DataFrame

c) Syntax to change single value-

```
>>> df
   One  Two  Three  Four
A  1.0  2.0   3.0  66.0
B  4.0  5.0   6.0  66.0
C  7.0  8.0   9.0  66.0
D 99.0 99.0  99.0  99.0
```

```
>>> df['Four']=[10,11,12,13]
```

```
>>> df.at['D']=[13,14,15,16]
```

```
>>> df
   One  Two  Three  Four
A  1.0  2.0   3.0  10.0
B  4.0  5.0   6.0  11.0
C  7.0  8.0   9.0  12.0
D 13.0 14.0  15.0  16.0
```

```
>>> df.Three['D']=100
```

```
>>> df
   One  Two  Three  Four
A  1.0  2.0   3.0  66.0
B  4.0  5.0   6.0  66.0
C  7.0  8.0   9.0  66.0
D 99.0 99.0  100.0 99.0
```

Here, value of column ‘Three’ of row ‘D’ got changed.

Values can be changed like this also. Values of row and column can be given separately.
Accessing and modifying values in DataFrame

d) Syntax for Column deletion-

\[
\text{del} <\text{DFObject}>[[<\text{ColName}>]\] \quad \text{or} \\
\text{df}.\text{drop}([[<\text{Col1Name}>,<\text{Col2Name}>], \ldots], \text{axis}=1)
\]

```python
>>> del df['Four']
>>> df
   One  Two  Three
A  1.0  2.0   3.0
B  4.0  5.0   6.0
C  7.0  8.0   9.0
D 13.0 14.0  15.0
>>> df.drop(["Two","Three"],axis=1)
   One
A  1.0
B  4.0
C  7.0
D 13.0
```

- `axis = 1` specifies deletion of column.
- `del` command does not return value after deletion whereas `drop` method returns the value to dataframe after deletion.

Neha Tyagi, KV5 Jaipur, II Shift
Iteration in DataFrame

• Sometimes we need to perform iteration on complete DataFrame. In such cases, it is difficult to write code to access values separately. Therefore, it is necessary to perform iteration on dataframe which is to be done as:

• `<DFObj>.iterrows()` it represents dataframe in row-wise subsets.

• `<DFObj>.iteritems()` it represents dataframe in column-wise subsets.
Use of pandas.iterrows () function

```python
import pandas as pd

disales={2015:{'Qtr1':34500,'Qtr2':56000,'Qtr3':47000,'Qtr4':49000},
         2016:{'Qtr1':44500,'Qtr2':46100,'Qtr3':57000,'Qtr4':59000},
         2017:{'Qtr1':54500,'Qtr2':51000,'Qtr3':47000,'Qtr4':58500}}

df1=pd.DataFrame(disales)

for (row,rowSeries) in df1.iterrows():
    print("RowIndex : ",row)
    print("Containing : ")
    print(rowSeries)

>>> df1
2015 2016 2017
Qtr1 34500 44500 54500
Qtr2 56000 46100 51000
Qtr3 47000 57000 47000
Qtr4 49000 59000 58500

These are the values of df1 which are processed one by one.

Try the code given below after creation of DataFrame.

```
Use of pandas.iteritems() function

```python
import pandas as pd

disales={
2015: {'Qtr1': 34500, 'Qtr2': 56000, 'Qtr3': 47000, 'Qtr4': 49000},
2016: {'Qtr1': 44500, 'Qtr2': 46100, 'Qtr3': 57000, 'Qtr4': 59000},
2017: {'Qtr1': 54500, 'Qtr2': 51000, 'Qtr3': 47000, 'Qtr4': 58500}
}

df1 = pd.DataFrame(disales)
for (col, colSeries) in df1.items():
    print("Column Index : ", col)
    print("Containing : ")
    print(colSeries)

>>> df1
          2015  2016  2017
Qtr1    34500  44500  54500
Qtr2    56000  46100  51000
Qtr3    47000  57000  47000
Qtr4    49000  59000  58500

These are the values of df1 which are processed one by one.

Try the code given below after creation of DataFrame.

```python
df1 = pd.DataFrame(disales)
for (col, colSeries) in df1.items():
    print("Column Index : ", col)
    print("Containing : ")
    i = 0
    for val in colSeries:
        print("At row", i, ": ", val)
        i = i + 1
```
Program for iteration

- Write a program to iterate over a dataframe containing names and marks, then calculates grades as per marks (as per guideline below) and adds them to the grade column.

  - Marks $\geq 90$ Grade A+
  - Marks 70 – 90 Grade A
  - Marks 60 – 70 Grade B
  - Marks 50 – 60 Grade C
  - Marks 40 – 50 Grade D
  - Marks $< 40$ Grade F
import pandas as pd
import numpy as np

names = pd.Series(['Sanjeev', 'Rajeev', 'Sanjay', 'Abhay'])
marks = pd.Series([76, 86, 55, 54])
stud = {'Name': names, 'Marks': marks}
df = pd.DataFrame(stud, columns=['Name', 'Marks'])
df['Grade'] = np.nan  # this will add NaN to all records of dataframe

print("Initial values in DataFrame")
print(df)

for (col, colSeries) in df.iteritems():
    length = len(colSeries)
    if col == 'Marks':
        lstMrks = []
        for row in range(length):
            mrks = colSeries[row]
            if mrks >= 90:
                lstMrks.append('A+')
            elif mrks >= 70:
                lstMrks.append('A')
            elif mrks >= 60:
                lstMrks.append('B')
            elif mrks >= 50:
                lstMrks.append('C')
            elif mrks >= 40:
                lstMrks.append('D')
            else:
                lstMrks.append('F')

    df['Grade'] = lstMrks

print("\nDataFrame after calculation of Grades")
print(df)
Binary Operations in a DataFrame

It is possible to perform add, subtract, multiply and division operations on DataFrame.

To Add - ( +, add or radd )
To Subtract - (-, sub or rsub)
To Multiply– ( * or mul)
To Divide - (/ or div)

We will perform operations on following dataframes-

<table>
<thead>
<tr>
<th>df1</th>
<th>df2</th>
<th>df3</th>
<th>df4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>40</td>
<td>300</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>90</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>300</td>
<td>4000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>90</td>
<td>6000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>4000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80</td>
<td>6000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>90</td>
<td>6000</td>
</tr>
</tbody>
</table>
DataFrame follows index matching to perform arithmetic operations. If matches, operation takes place otherwise it shows NaN (Not a Number). It is called *Data Alignment* in panda object. This behavior of ‘data alignment’ on the basis of “matching indexes” is called MATCHING.
Subtraction

Neha Tyagi, KV5 Jaipur, II Shift

```
>>> df1
   A  B  C
0  0  1  2  3
1  1  4  5  6
2  2  7  8  9

>>> df2
   A  B  C
0  0  10 20 30
1  1  40 50 60
2  2  70 80 90

>>> df3
   A  B  C
0  0  100 200 300
1  1  400 500 600

>>> df4
   A  B
0  0  1000 2000
1  1  3000 4000
2  2  5000 6000
```

```
>>> df1-df2
   A  B  C
0 -9 -18 -27
1 -36 -45 -54
2 -63 -72 -81

>>> df1-df3
   A  B  C
0 -99.0 -198.0 -297.0
1 -396.0 -495.0 -594.0
2 NaN NaN NaN

>>> df1-df4
   A  B  C
0 -999 -1998 NaN
1 -2996 -3995 NaN
2 -4993 -5992 NaN
```

```
>>> df1.sub(df2)
   A  B  C
0 -9 -18 -27
1 -36 -45 -54
2 -63 -72 -81

>>> df1.sub(df3)
   A  B  C
0 -99.0 -198.0 -297.0
1 -396.0 -495.0 -594.0
2 NaN NaN NaN

>>> df1.sub(df4)
   A  B  C
0 -999 -1998 NaN
1 -2996 -3995 NaN
2 -4993 -5992 NaN
```

```
>>> df1.rsub(df2)
   A  B  C
0  9  18  27
1  36  45  54
2  63  72  81

>>> df1.rsub(df3)
   A  B  C
0  99.0  198.0  297.0
1  396.0  495.0  594.0
2 NaN NaN NaN

>>> df1.rsub(df4)
   A  B  C
0  999  1998 NaN
1  2996  3995 NaN
2  4993  5992 NaN
```
### Multiplication

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>1</td>
<td>3000</td>
<td>4000</td>
</tr>
<tr>
<td>2</td>
<td>5000</td>
<td>6000</td>
</tr>
</tbody>
</table>

```python
>>> df1
A   B   C
0   1   2   3
1   4   5   6
2   7   8   9

>>> df2
A   B   C
0  10  20  30
1  40  50  60
2  70  80  90

>>> df3
A   B   C
0  100 200 300
1  400 500 600

>>> df4
A   B   C
0  1000 2000
1  3000 4000
2  5000 6000
```

```python
>>> df1*df2
A   B   C
0   10  40  90
1  160 250 360
2  490 640 810

>>> df1*df3
A   B   C
0  100.0 400.0 900.0
1 1600.0 2500.0 3600.0
2   NaN   NaN   NaN

>>> df1*df4
A   B   C
0  1000 4000   NaN
1 12000 20000  NaN
2 35000 48000  NaN
```

```python
>>> df1.mul(df2)
A   B   C
0   10  40  90
1  160 250 360
2  490 640 810

>>> df1.mul(df3)
A   B   C
0  100.0 400.0 900.0
1 1600.0 2500.0 3600.0
2   NaN   NaN   NaN

>>> df1.mul(df4)
A   B   C
0  1000 4000   NaN
1 12000 20000  NaN
2 35000 48000  NaN
```
### Division

<table>
<thead>
<tr>
<th>df1</th>
<th>df2</th>
<th>df3</th>
<th>df4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>500</td>
<td>600</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>600</td>
<td></td>
</tr>
</tbody>
</table>

#### df1 / df2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

#### df2 / df1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

#### df1 / df3

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

#### df3 / df1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

#### df1 / df2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

#### df2 / df1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>1</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>2</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>

#### df1 / df3

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

#### df3 / df1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

#### df1 / df2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

#### df2 / df1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>1</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>2</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>

#### df1 / df3

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

#### df3 / df1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>1</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

See the operation of the `rdiv` carefully.
Other important functions

Other important functions of DataFrame are as under-

```python
<DF>.info()
```

```python
>>> df1
   A  B  C
0  0  1  2
1  1  4  5
2  2  7  8
```

```python
<DF>.describe()
```

```python
>>> df1.describe()
   A    B    C
count 3.00 3.00 3.00
mean  4.00  5.00  6.00
std   3.00  3.00  3.00
min   1.00  2.00  3.00
25%   2.50  3.50  4.50
50%   4.00  5.00  6.00
75%   5.50  6.50  7.50
max   7.00  8.00  9.00
```
Other important functions of DataFrame are as under-

**<DF>.head ([ n=<n> ] )** here, default value of n is 5.

**<DF>.tail ( [n=<n>] )**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>500</td>
<td>600</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
<td>800</td>
<td>900</td>
</tr>
</tbody>
</table>

**<DF>.head(n=3)**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

**<DF>.tail(n=4)**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>70</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>500</td>
<td>600</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
<td>800</td>
<td>900</td>
</tr>
</tbody>
</table>

Neha Tyagi, KV5 Jaipur, II Shift
Cumulative Calculations Functions

In DataFrame, for cumulative sum, function is as under-

```python
>>> df1.cumsum([axis = None])
```

here, axis argument is optional.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```python
>>> df1.cumsum(axis='rows')
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

```python
>>> df1.cumsum(axis='columns')
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>15</td>
</tr>
</tbody>
</table>

Neha Tyagi, KV5 Jaipur, II Shift
Index of Maximum and Minimum Values

```python
>>> df5
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
3 10 20 30
4 40 50 60
5 70 80 90
6 100 200 300
7 400 500 600
8 700 800 900

>>> df5.idxmax()
A     8
B     8
C     8
dtype: int64

>>> df5.idxmin()
A     0
B     0
C     0
dtype: int64
```
Handling of Missing Data

• The values with no computational significance are called missing values.

• Handling methods for missing values-
   Dropping missing data
   Filling missing data (Imputation)

```
>>> df10
   A   B   C
0  1001  2002  NaN
1  3004  4005  NaN
2  5007  6008  NaN

>>> df11=df10.dropna()  # Dropping missing data
>>> df11
Empty DataFrame
Columns: [A, B, C]
Index: []

>>> df11.fillna(0)  # Filling missing data with 0
>>> df11
   A   B   C
0  1001  2002   0.0
1  3004  4005   0.0
2  5007  6008   0.0
```
### Comparison of Pandas Objects

```python
>>> df1
  A  B  C
0  1  2  3
1  4  5  6
2  7  8  9

>>> df2
  A  B  C
0  0  1  0
1  1  2  0
2  2  3  0

>>> df3
  A  B  C
0  0  1  0
1  4  0  0
2  NaN NaN NaN

>>> df12
  A  B  C
0  101 202 303
1  404 505 606
2  NaN NaN NaN

>>> df1+df2==df1.add(df2)
  A  B  C
0  True True True
1  True True True
2  True True True

>>> df1+df3==df1.add(df3)
  A  B  C
0  True True True
1  True True True
2  False False False

>>> (df1+df3).equals(df1.add(df3))
True
```

equals () checks both the objects for equality.
• कृपया हमारे ब्लॉग को फॉलो करिए और youtube channel को subscribe करिए | ताकक आपको और सारे chapters मिल सकें |

www.pythontrends.wordpress.com