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

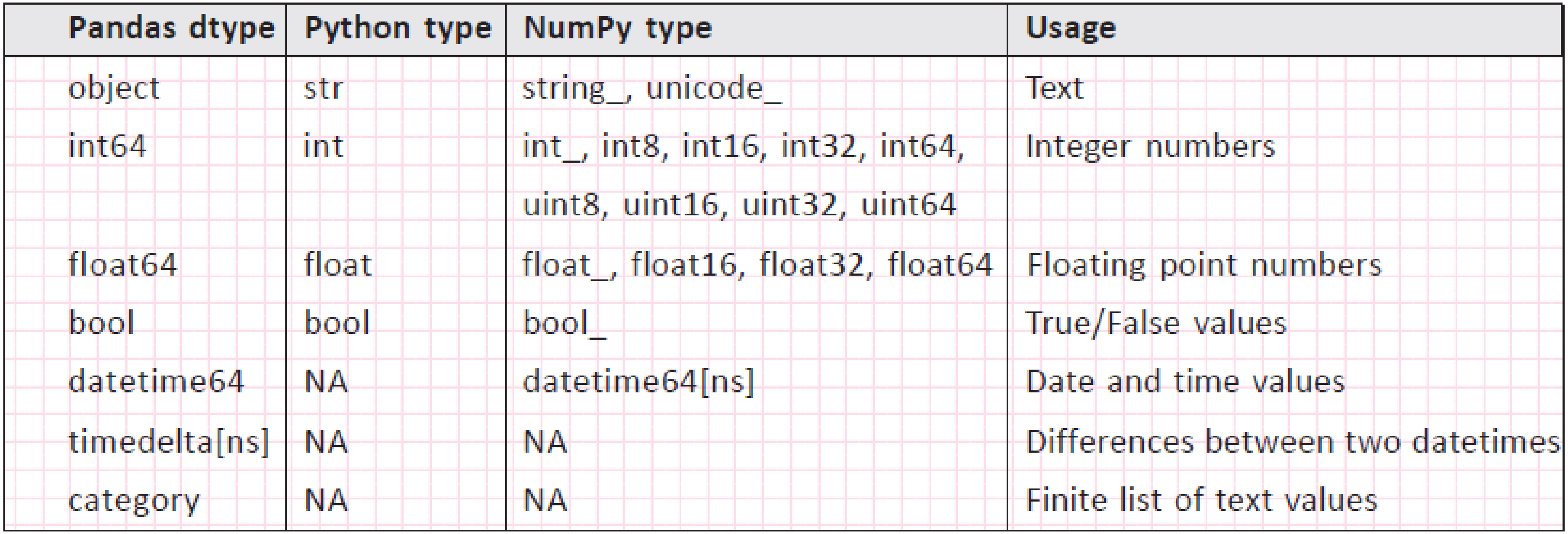
Jaipur Region

Python Pandas (A Review)

- Data Processing is the most important part of Data Analysis. Because data is not avaiable 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

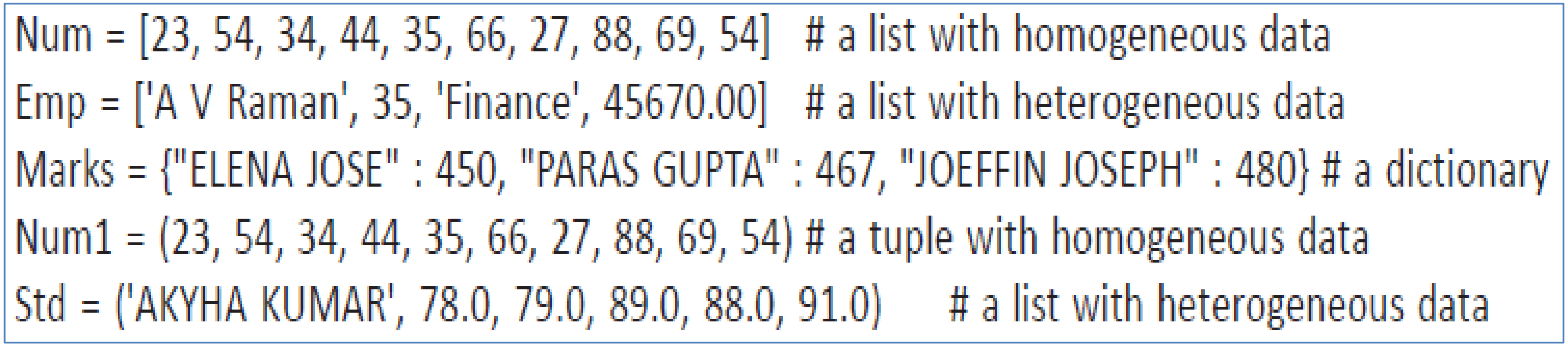
- Pandas is an open-source library of python providing high-performance data manipulation and analysis tool using its powerful data structure.
- Pandas provides rich set of functions to process various types of data.
- During data analysis it is very important to make it confirm that you are using correct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



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

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* .
- Lets take some data which can be considered as series -



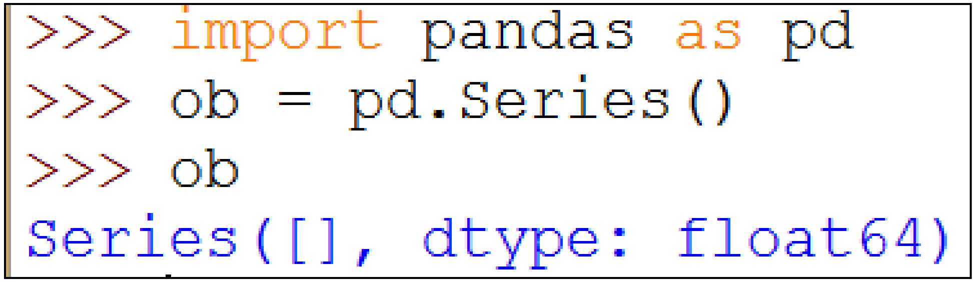
```
Num = [23, 54, 34, 44, 35, 66, 27, 88, 69, 54]   # a list with homogeneous data
Emp = ['A V Raman', 35, 'Finance', 45670.00]   # a list with heterogeneous data
Marks = {"ELENA JOSE" : 450, "PARAS GUPTA" : 467, "JOEFFIN JOSEPH" : 480} # a dictionary
Num1 = (23, 54, 34, 44, 35, 66, 27, 88, 69, 54) # 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.



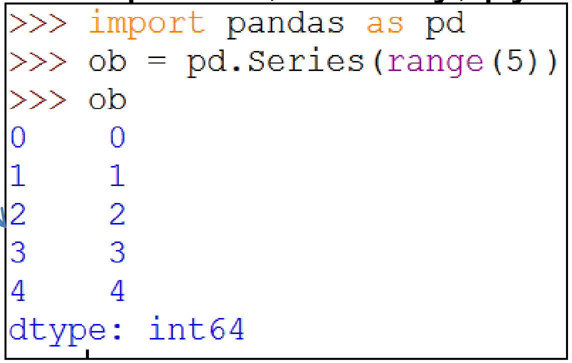
```
>>> import pandas as pd
>>> ob = pd.Series()
>>> ob
Series([], dtype: float64)
```

2. Non-empty series creation–

Import pandas as pd

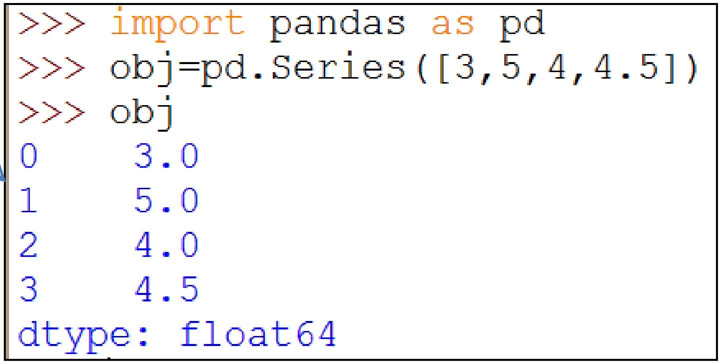
<Series Object> = pd.Series(data, index=idx) where data can be python sequence, ndarray, python dictionary or scaler value.

Index



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

Index

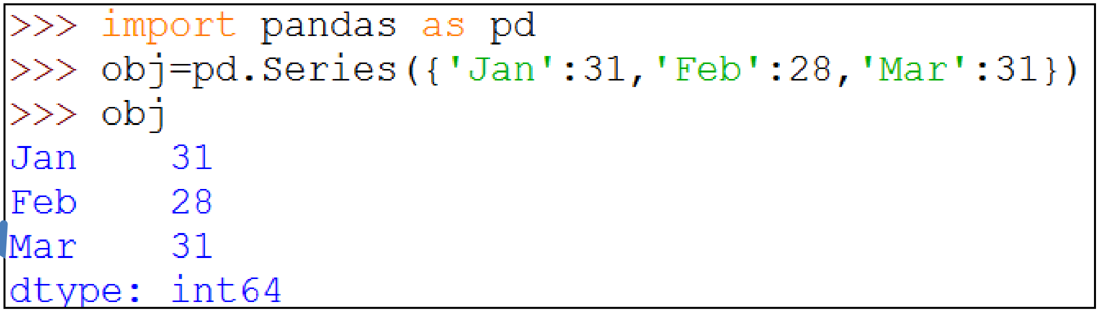


```
>>> import pandas as pd
>>> obj=pd.Series([3,5,4,4.5])
>>> obj
0     3.0
1     5.0
2     4.0
3     4.5
dtype: float64
```

Series Objects creation

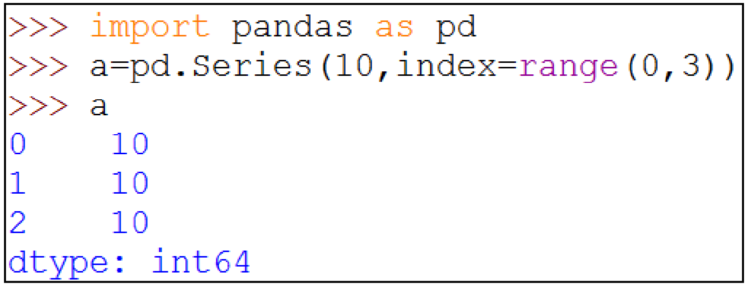
1. Creation of series with Dictionary-

Index of Keys

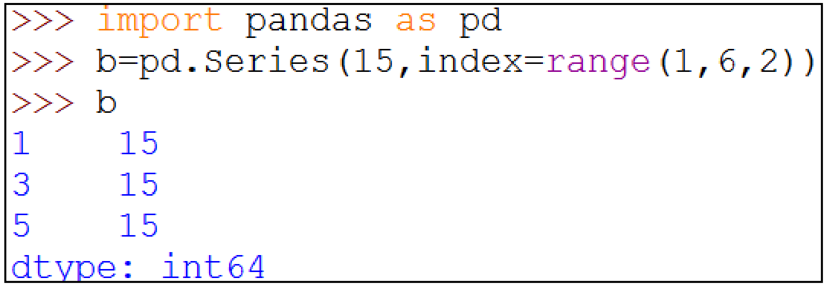


```
>>> import pandas as pd
>>> obj=pd.Series({'Jan':31,'Feb':28,'Mar':31})
>>> obj
Jan    31
Feb    28
Mar    31
dtype: int64
```

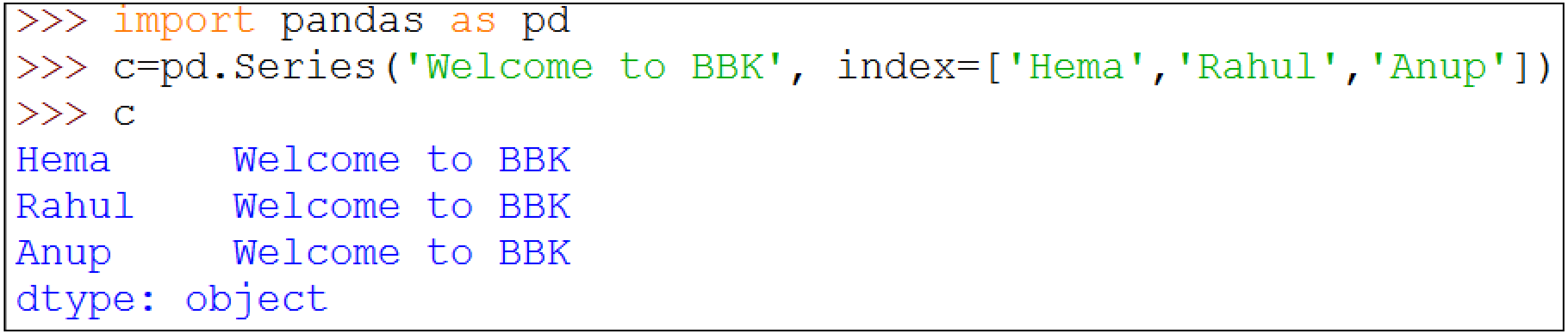
2. Creation of series with Scalar value-



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



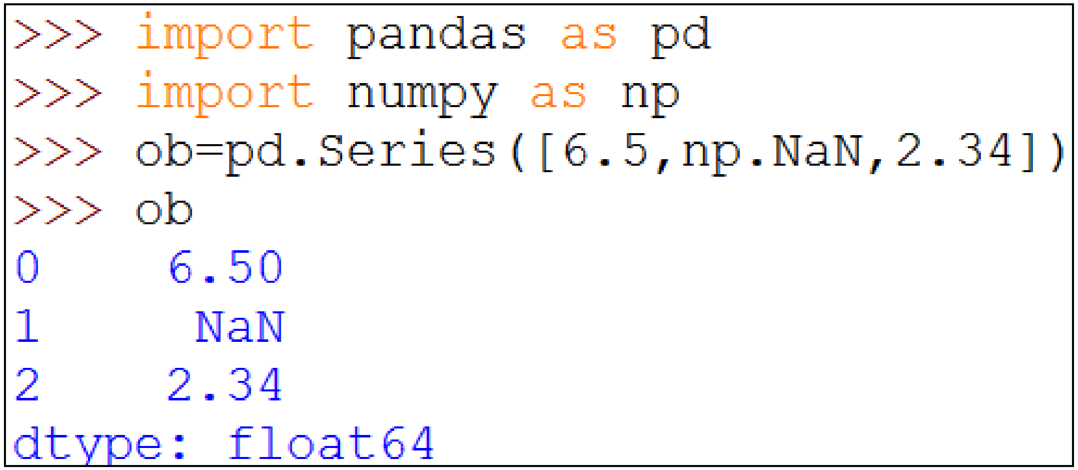
```
>>> import pandas as pd
>>> b=pd.Series(15,index=range(1,6,2))
>>> b
1    15
3    15
5    15
dtype: int64
```



```
>>> 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
```

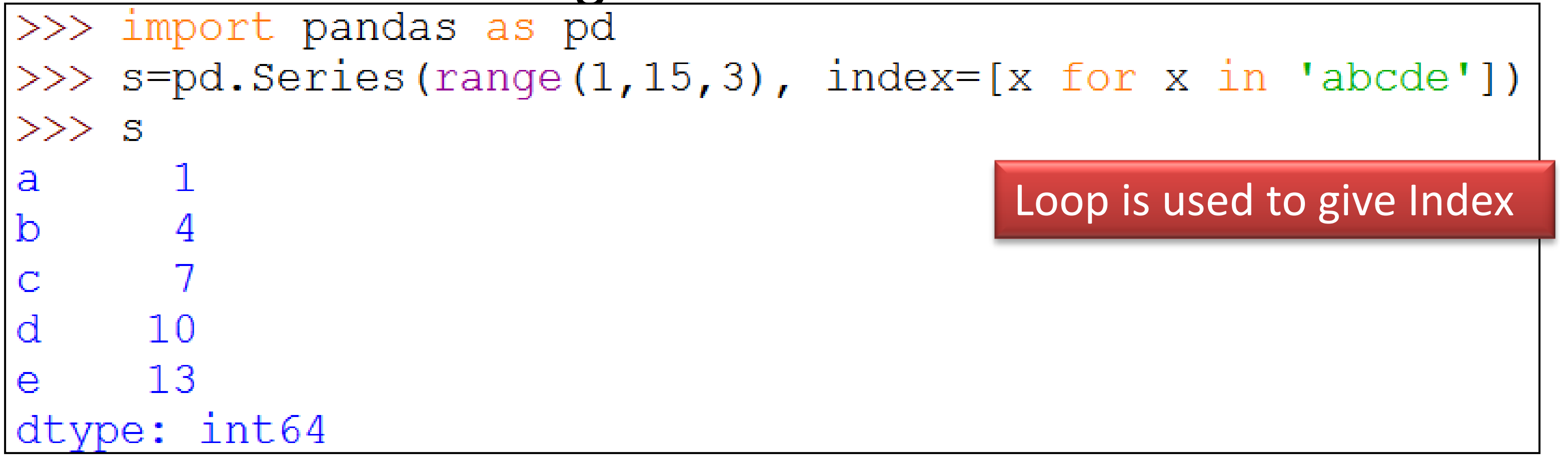
Creation of Series Objects –Additional functionality

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.



```
>>> 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-

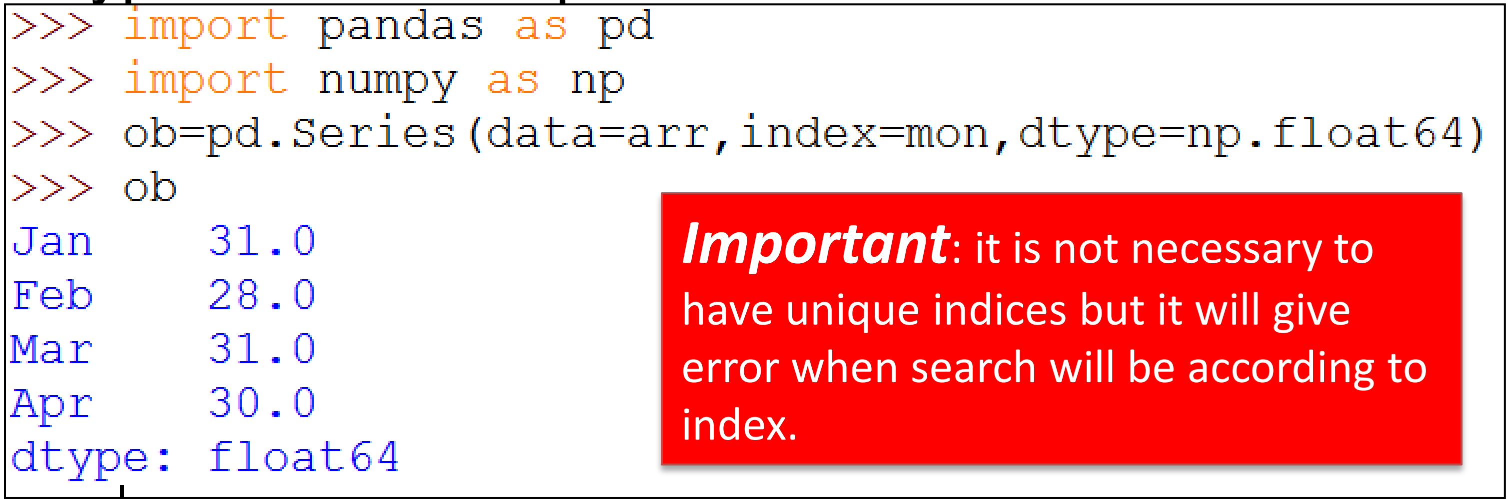


```
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3), index=[x for x in 'abcde'])
>>> s
a     1
b     4
c     7
d    10
e    13
dtype: int64
```

Loop is used to give Index

Creation of Series Objects –Additional functionality

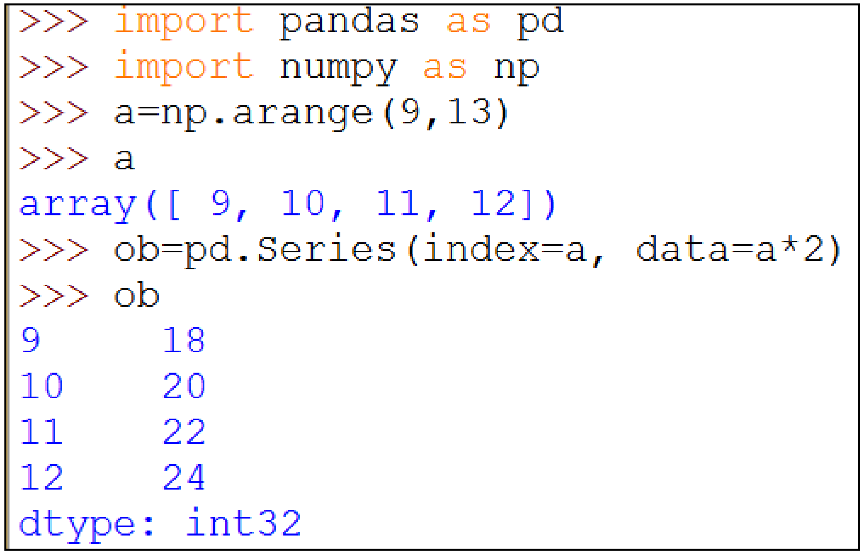
3. Dtype can also be passed with Data and index



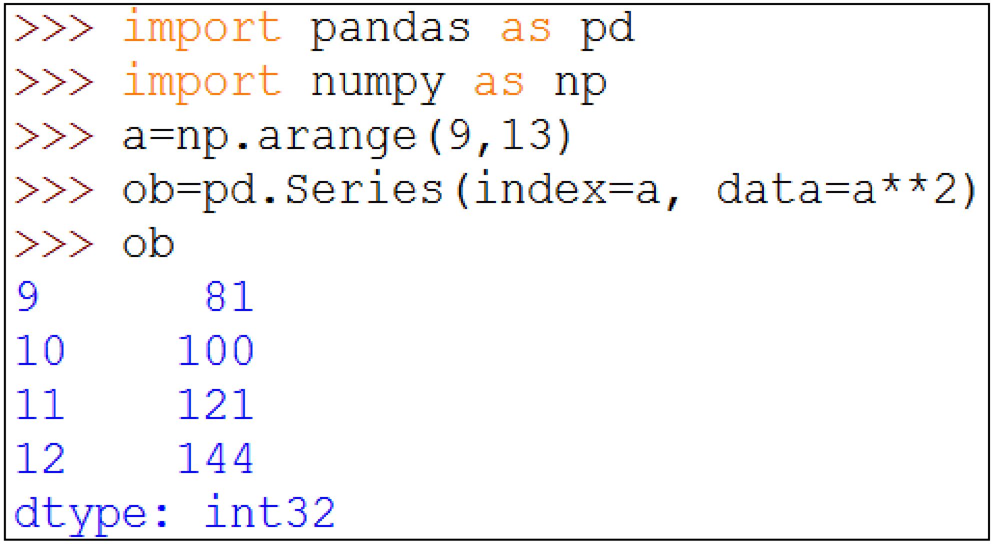
```
>>> import pandas as pd
>>> import numpy as np
>>> ob=pd.Series(data=arr,index=mon,dtype=np.float64)
>>> ob
Jan    31.0
Feb    28.0
Mar    31.0
Apr    30.0
dtype: float64
```

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-



```
>>> 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     18
10    20
11    22
12    24
dtype: int32
```



```
>>> import pandas as pd
>>> import numpy as np
>>> a=np.arange(9,13)
>>> ob=pd.Series(index=a, data=a**2)
>>> ob
9      81
10    100
11    121
12    144
dtype: int32
```

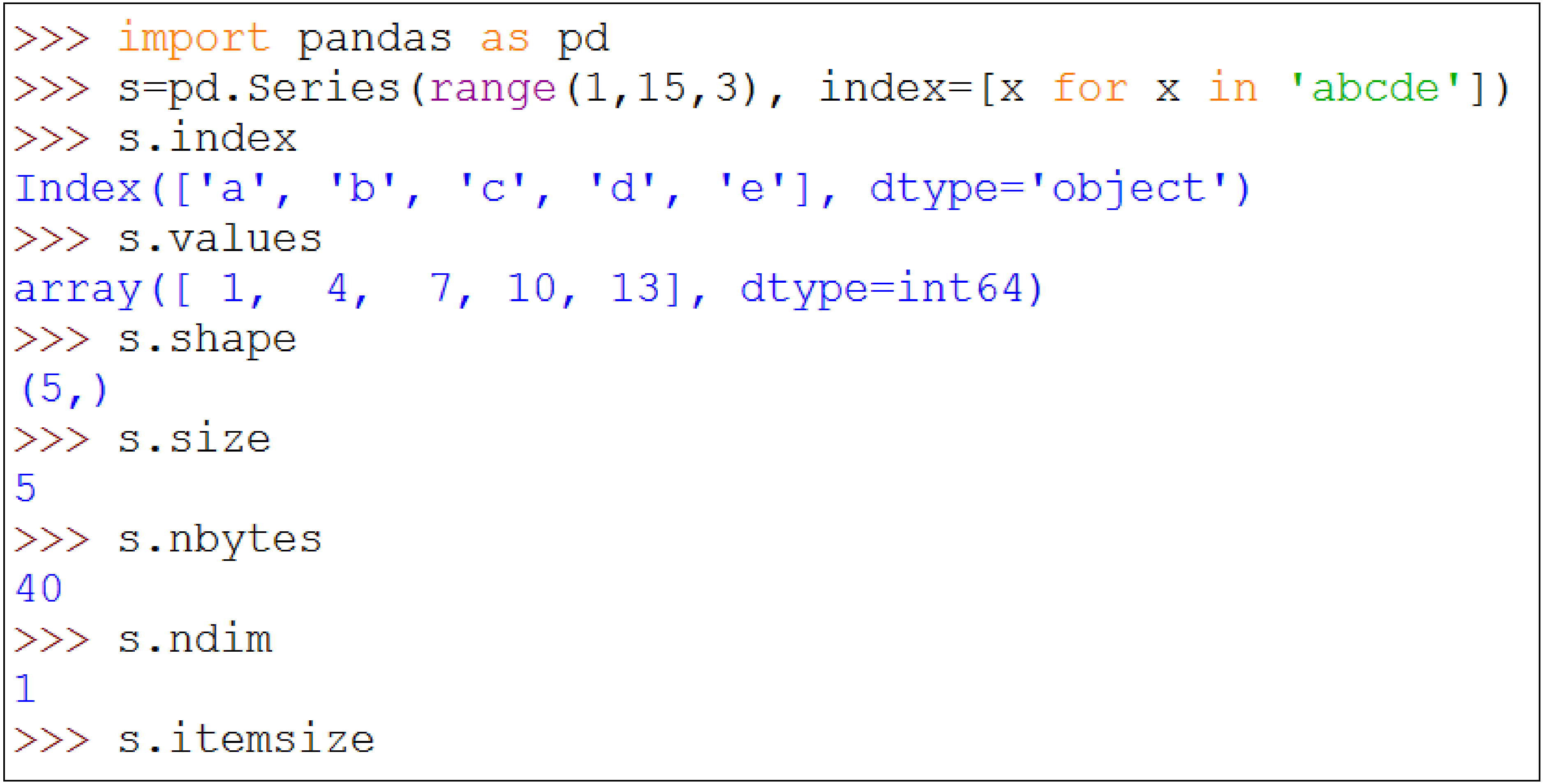
Series Object Attributes

3. Some common attributes-

<series object>.<AttributeName>

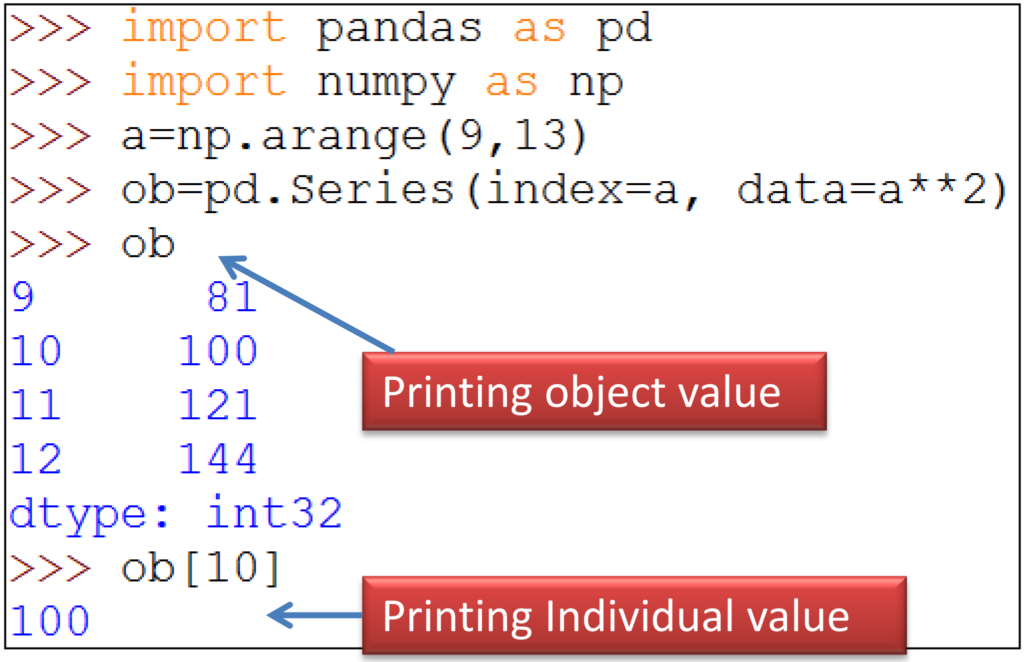
Attribute	Description
Series.index	Returns index of the series
Series.values	Returns ndarray
Series.dtype	Returns dtype object of the underlying data
Series.shape	Returns tuple of the shape of underlying data
Series.nbytes	Return number of bytes of underlying data
Series.ndim	Returns the number of dimention
Series.size	Returns number of elements
Series.intemsize	Returns the size of the dtype
Series.hasnans	Returns true if there are any NaN
Series.empty	Returns true if series object is empty

Series Object Attributes



```
>>> 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
```

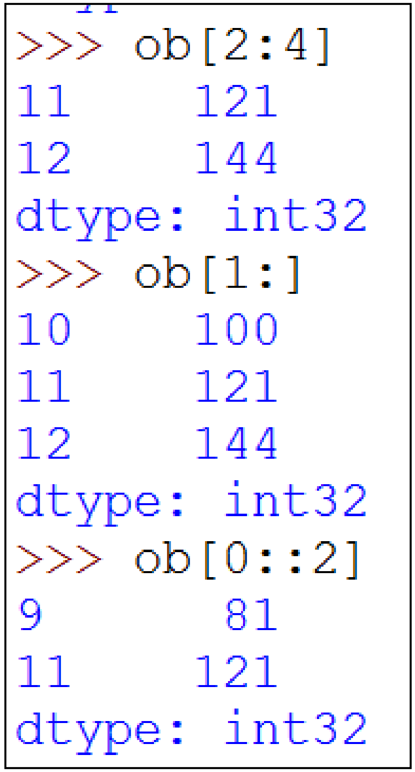
Accessing Series Object



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

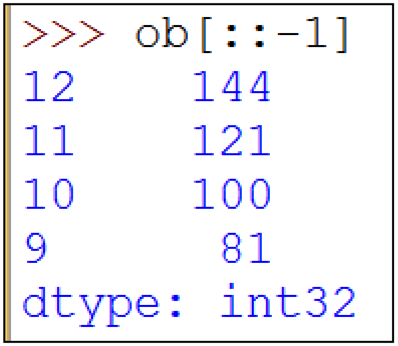
Printing object value

Printing Individual value



```
>>> ob[2:4]
11    121
12    144
dtype: int32
>>> ob[1:]
10    100
11    121
12    144
dtype: int32
>>> ob[0::2]
9      81
11    121
dtype: int32
```

Object slicing



```
>>> ob[::-1]
12    144
11    121
10    100
9      81
dtype: int32
```

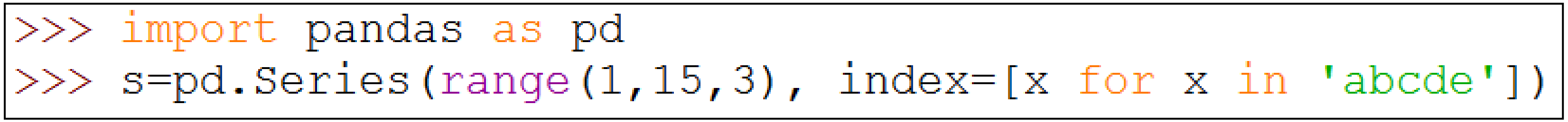
For Object slicing, follow the following syntax-

<objectName>[<start>:<stop>:<step >]

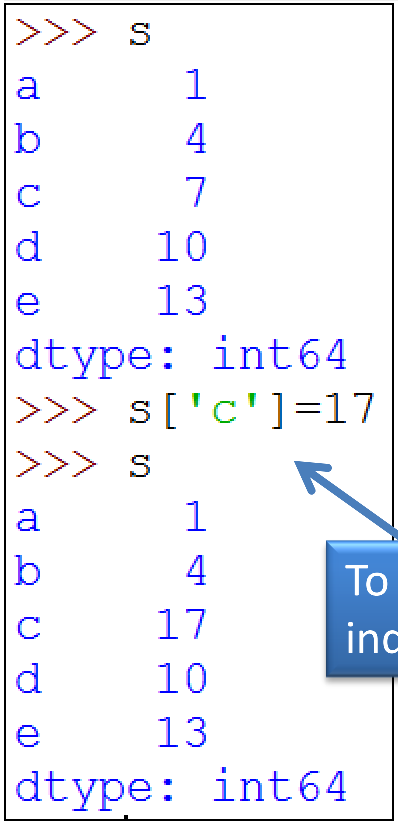
Operations on Series Object

1. Elements modification-

<series object>[index] = <new_data_value>

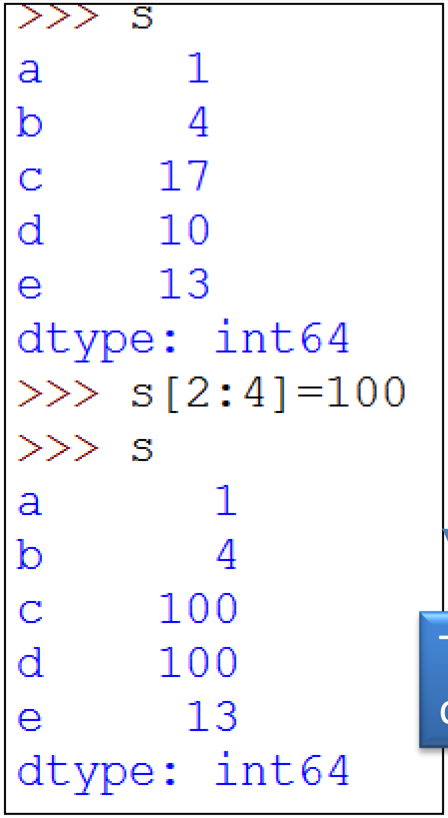


```
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3), index=[x for x in 'abcde'])
```

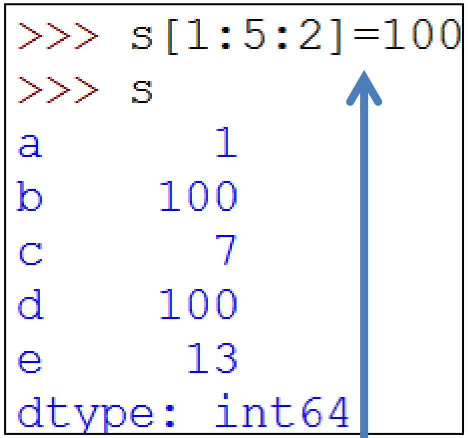


```
>>> s
a      1
b      4
c      7
d     10
e     13
dtype: int64
>>> s['c']=17
>>> s
a      1
b      4
c     17
d     10
e     13
dtype: int64
```

To change individual value



```
>>> s
a      1
b      4
c     17
d     10
e     13
dtype: int64
>>> s[2:4]=100
>>> s
a      1
b      4
c    100
d    100
e     13
dtype: int64
```



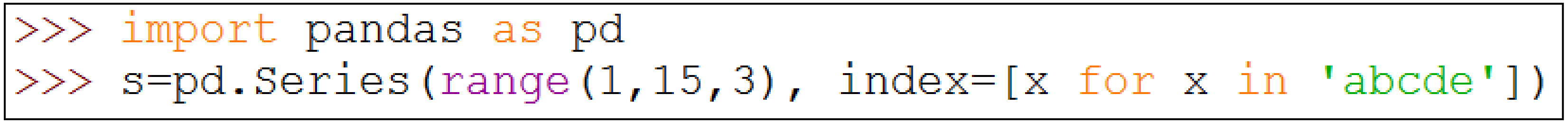
```
>>> s[1:5:2]=100
>>> s
a      1
b    100
c      7
d    100
e     13
dtype: int64
```

To change value in a certain slice

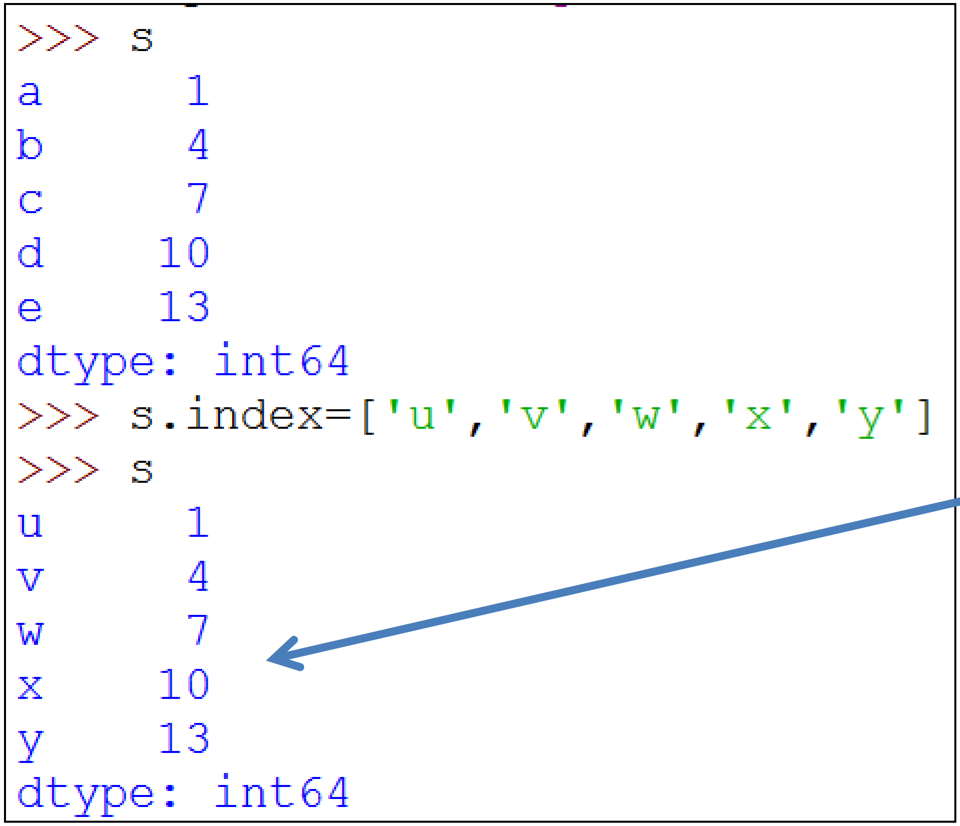
Operations on Series Object

1. It is possible to change indexes

<series object>.<index] = <new_index_array>



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

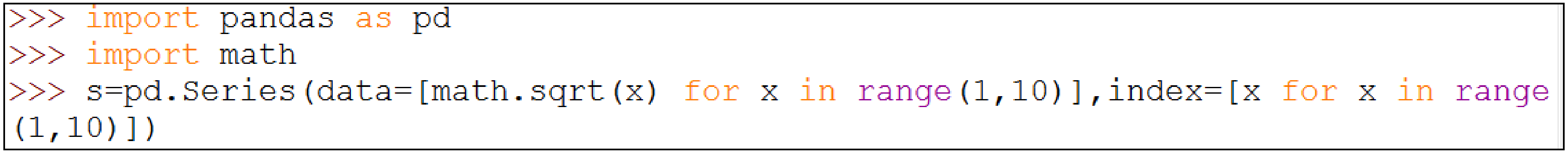


```
>>> s
a      1
b      4
c      7
d     10
e     13
dtype: int64
>>> s.index=['u','v','w','x','y']
>>> s
u      1
v      4
w      7
x     10
y     13
dtype: int64
```

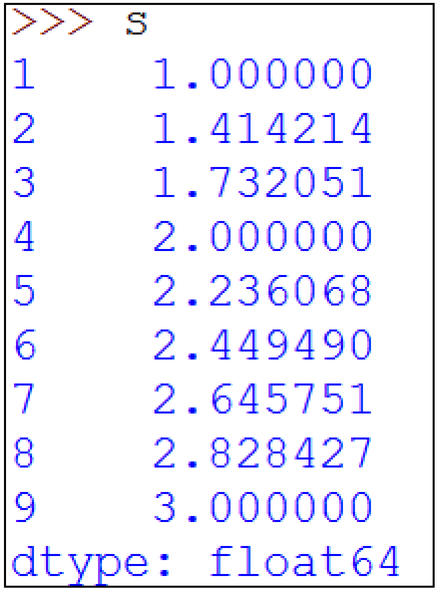
Here, indexes got changed.

head() and tail () Function

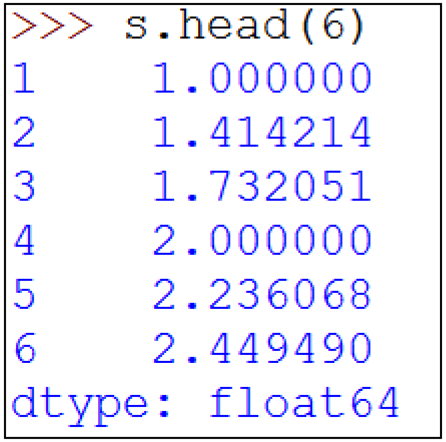
1. head(<n>) function fetch first n rows from a pandas object. If you do not provide any value for n, will return first 5 rows.
2. tail(<n>) function fetch last n rows from a pandas object. If you do not provide any value for n, will return last 5 rows.



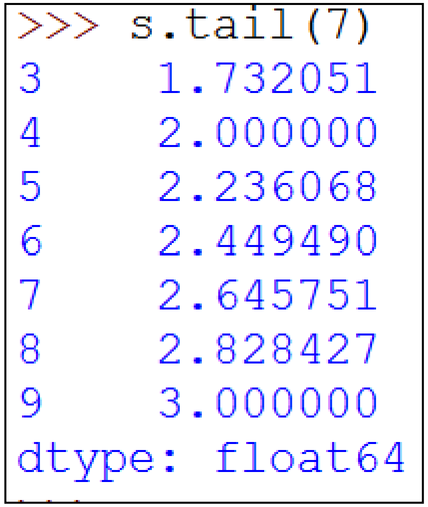
```
>>> import pandas as pd
>>> import math
>>> s=pd.Series(data=[math.sqrt(x) for x in range(1,10)],index=[x for x in range
(1,10)])
```



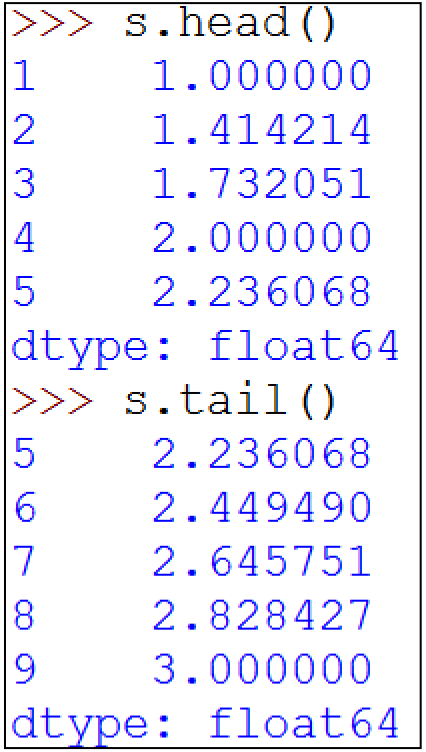
```
>>> s
1    1.000000
2    1.414214
3    1.732051
4    2.000000
5    2.236068
6    2.449490
7    2.645751
8    2.828427
9    3.000000
dtype: float64
```



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

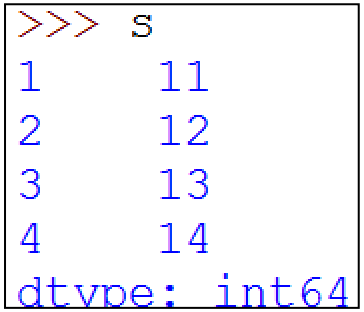


```
>>> s.tail(7)
3    1.732051
4    2.000000
5    2.236068
6    2.449490
7    2.645751
8    2.828427
9    3.000000
dtype: float64
```

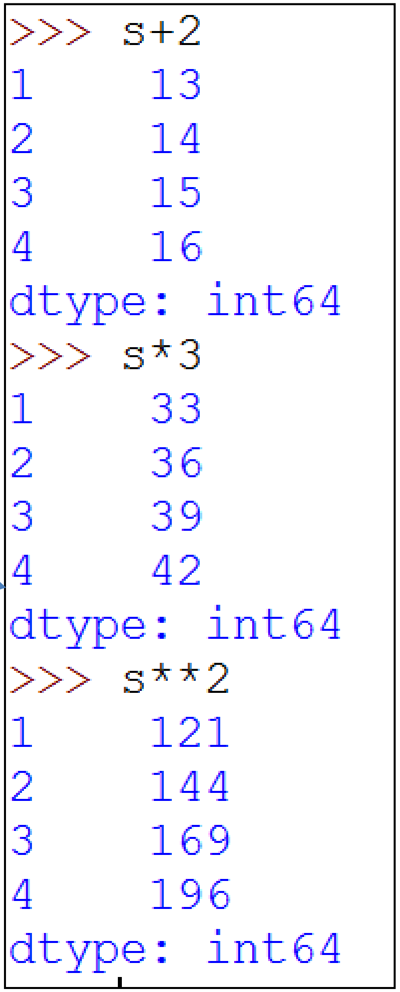


```
>>> s.head()
1    1.000000
2    1.414214
3    1.732051
4    2.000000
5    2.236068
dtype: float64
>>> s.tail()
5    2.236068
6    2.449490
7    2.645751
8    2.828427
9    3.000000
dtype: float64
```

Series Objects - Vector Operations

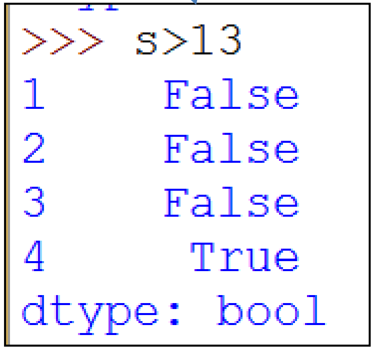


```
>>> s
1      11
2      12
3      13
4      14
dtype: int64
```



```
>>> 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
```

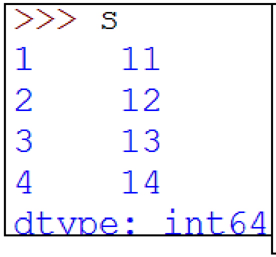
All these are vector operations



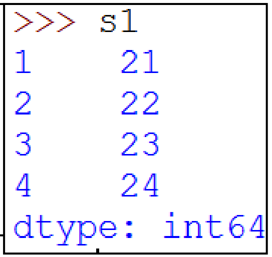
```
>>> s>13
1     False
2     False
3     False
4      True
dtype: bool
```

We can also store these results in other objects.

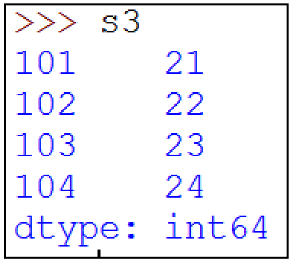
Series Objects - Arithmetic Operations



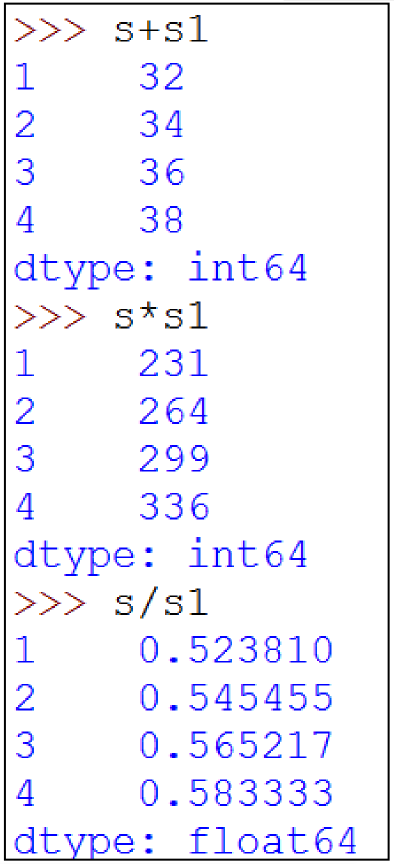
```
>>> s
1      11
2      12
3      13
4      14
dtype: int64
```



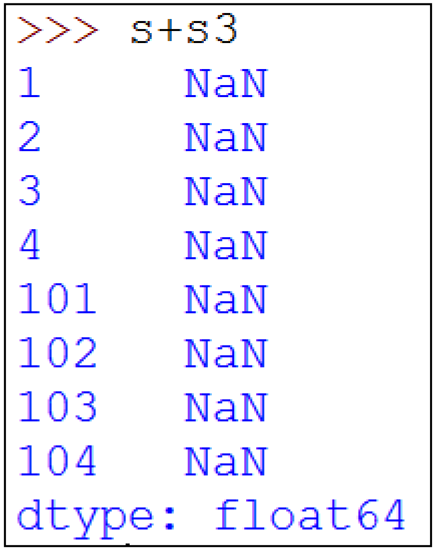
```
>>> s1
1      21
2      22
3      23
4      24
dtype: int64
```



```
>>> s3
101      21
102      22
103      23
104      24
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
```

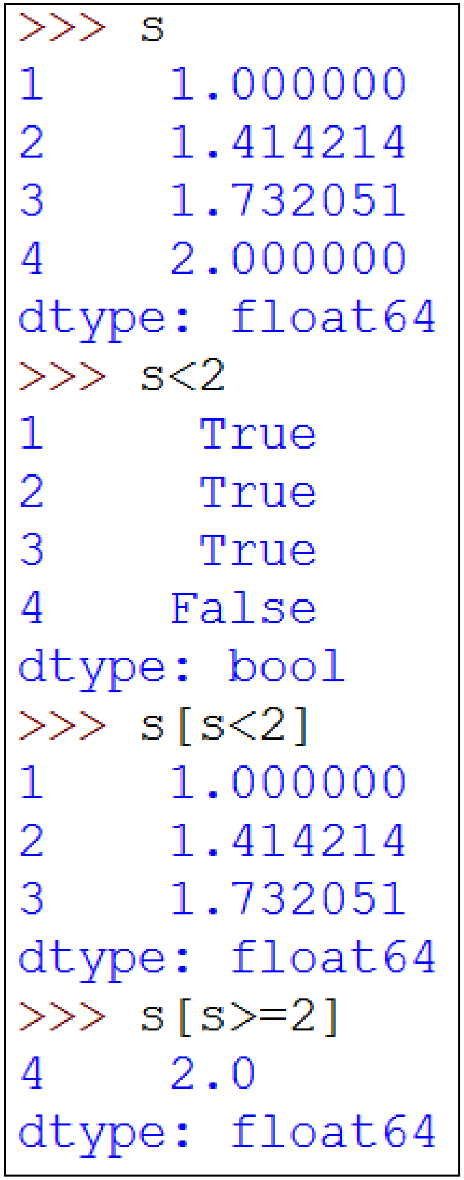


```
>>> s+s3
1       NaN
2       NaN
3       NaN
4       NaN
101     NaN
102     NaN
103     NaN
104     NaN
dtype: float64
```

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

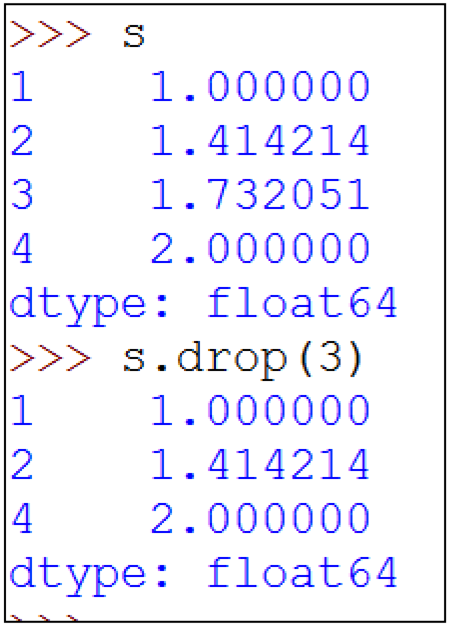
Entries Filtering

<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
```

Other feature

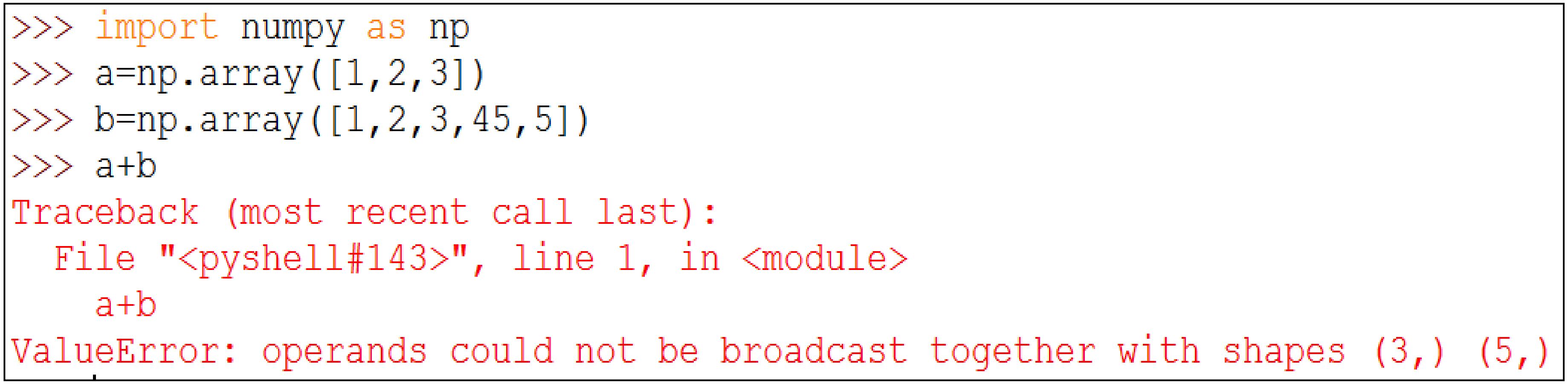


```
>>> s
1    1.000000
2    1.414214
3    1.732051
4    2.000000
dtype: float64
>>> s.drop(3)
1    1.000000
2    1.414214
4    2.000000
dtype: float64
```

To delete value of index

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.

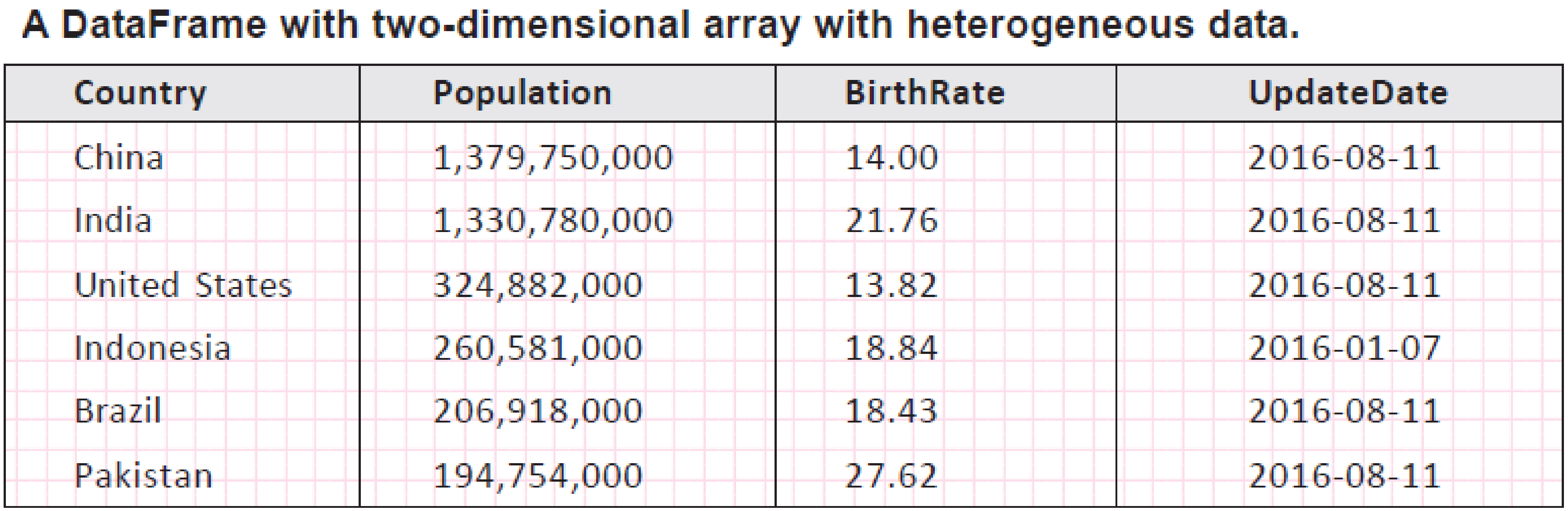


```
>>> import numpy as np
>>> a=np.array([1,2,3])
>>> b=np.array([1,2,3,45,5])
>>> a+b
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in <module>
    a+b
ValueError: operands could not be broadcast together with shapes (3,) (5,)
```

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.

Country	Population	BirthRate	UpdateDate
China	1,379,750,000	14.00	2016-08-11
India	1,330,780,000	21.76	2016-08-11
United States	324,882,000	13.82	2016-08-11
Indonesia	260,581,000	18.84	2016-01-07
Brazil	206,918,000	18.43	2016-08-11
Pakistan	194,754,000	27.62	2016-08-11

Creation and presentation of DataFrame

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

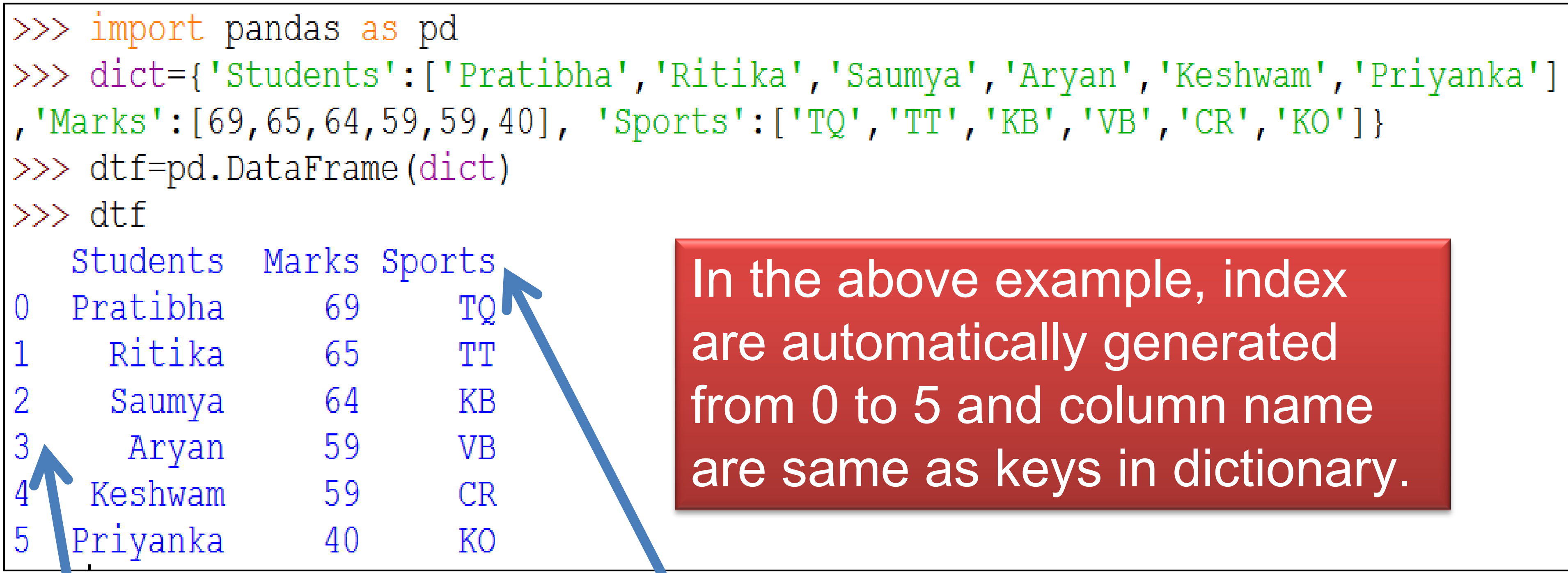
  `import pandas as pd`

```
<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.

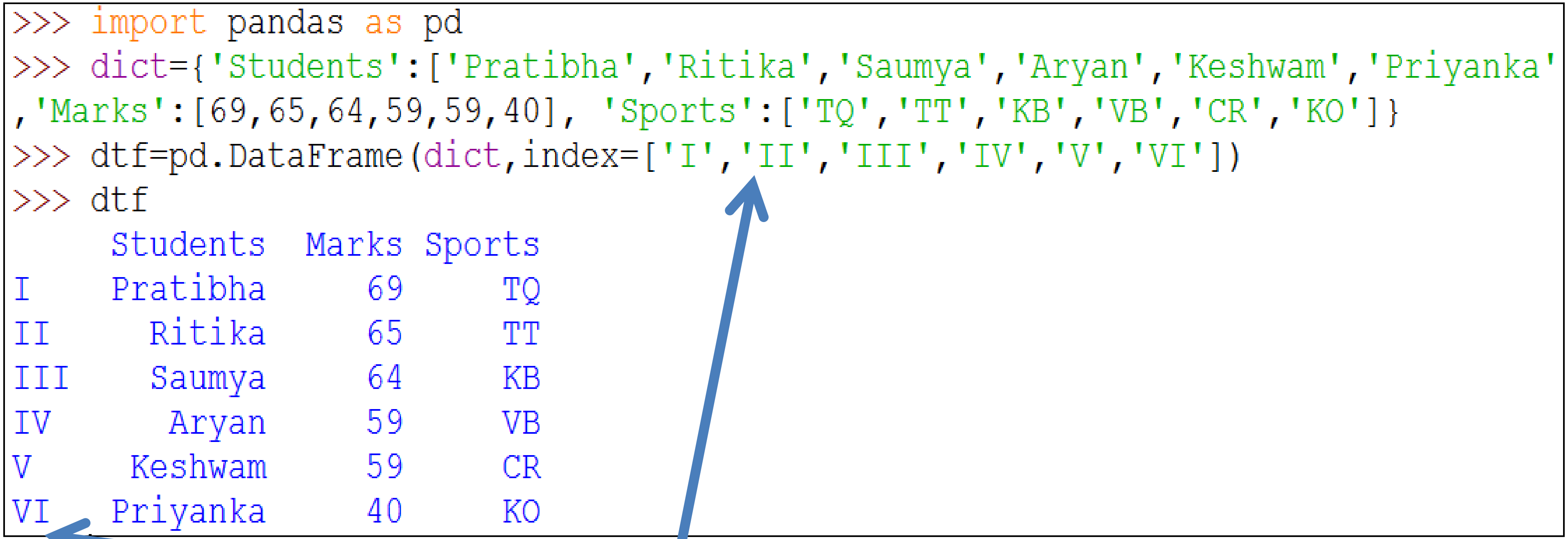


```
>>> import pandas as pd
>>> dict={'Students':['Pratibha','Ritika','Saumya','Aryan','Keshwam','Priyanka']
,'Marks':[69,65,64,59,59,40], 'Sports':['TQ','TT','KB','VB','CR','KO']}
>>> dtf=pd.DataFrame(dict)
>>> dtf
   Students  Marks Sports
0  Pratibha     69     TQ
1    Ritika     65     TT
2    Saumya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

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

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

column name are generated from keys of 2D Dictionary



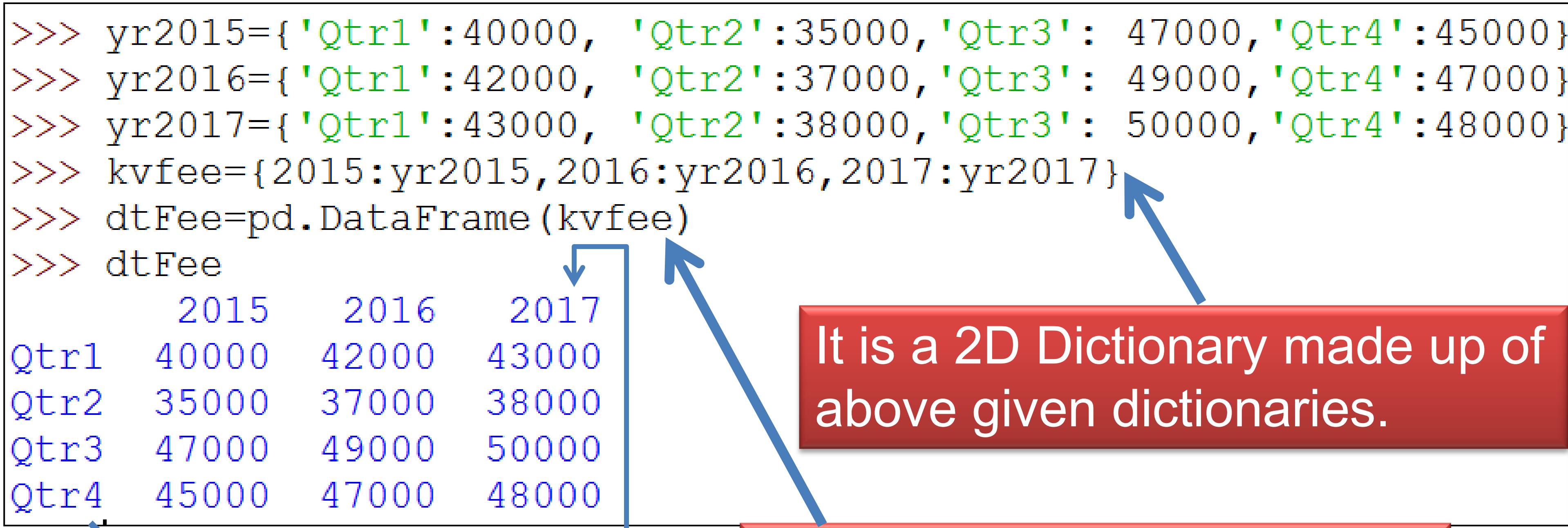
```
>>> import pandas as pd
>>> dict={'Students':['Pratibha','Ritika','Saumya','Aryan','Keshwam','Priyanka'
,'Marks':[69,65,64,59,59,40], 'Sports':['TQ','TT','KB','VB','CR','KO']}
>>> dtf=pd.DataFrame(dict,index=['I','II','III','IV','V','VI'])
>>> dtf
     Students  Marks Sports
I    Pratibha     69     TQ
II     Ritika     65     TT
III    Saumya     64     KB
IV      Aryan     59     VB
V     Keshwam     59     CR
VI   Priyanka     40     KO
```

Here, indexes are specified by you.

Meaning, if you specify the sequence of index then index wlil 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-



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

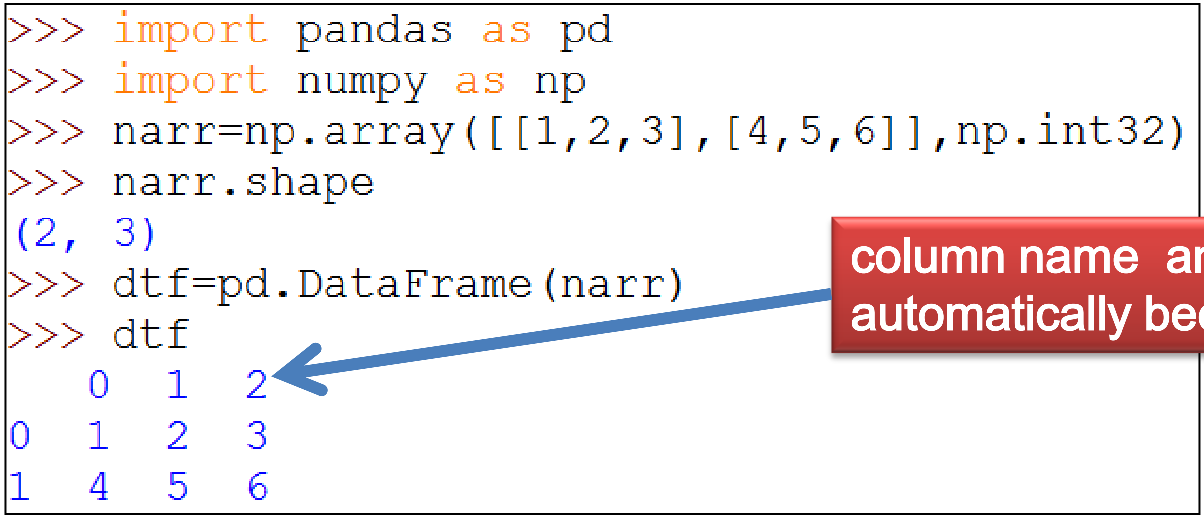
It is a 2D Dictionary made up of above given dictionaries.

DataFrame object created.

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

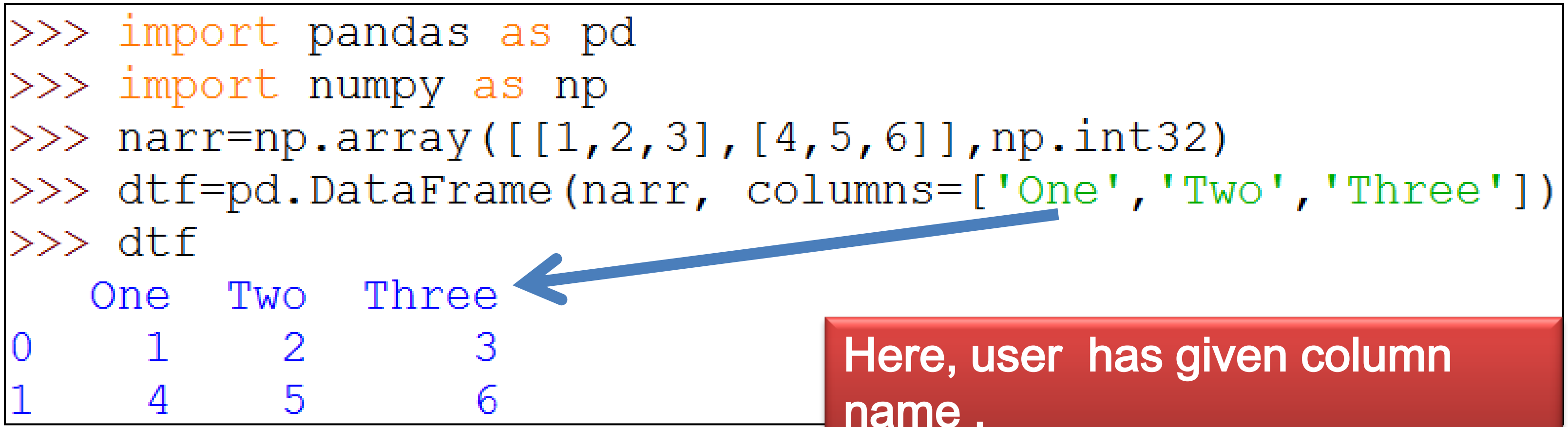
If keys of yr2015, yr2016 and yr2017 were different here then rows and columns of dataframe would have increased and non-matching rows and column would store NaN.

Creation of Dataframe from 2D ndarray



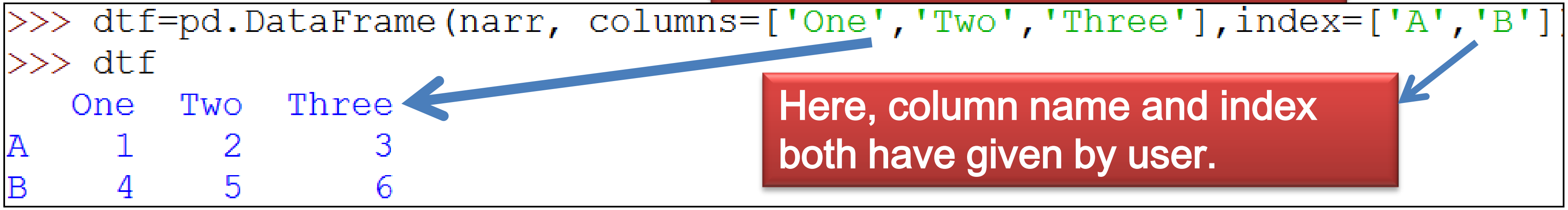
```
>>> import pandas as pd
>>> import numpy as np
>>> narr=np.array([[1,2,3],[4,5,6]],np.int32)
>>> narr.shape
(2, 3)
>>> dtf=pd.DataFrame(narr)
>>> dtf
   0  1  2
0  1  2  3
1  4  5  6
```

column name and index have automatically been generated here.



```
>>> 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
```

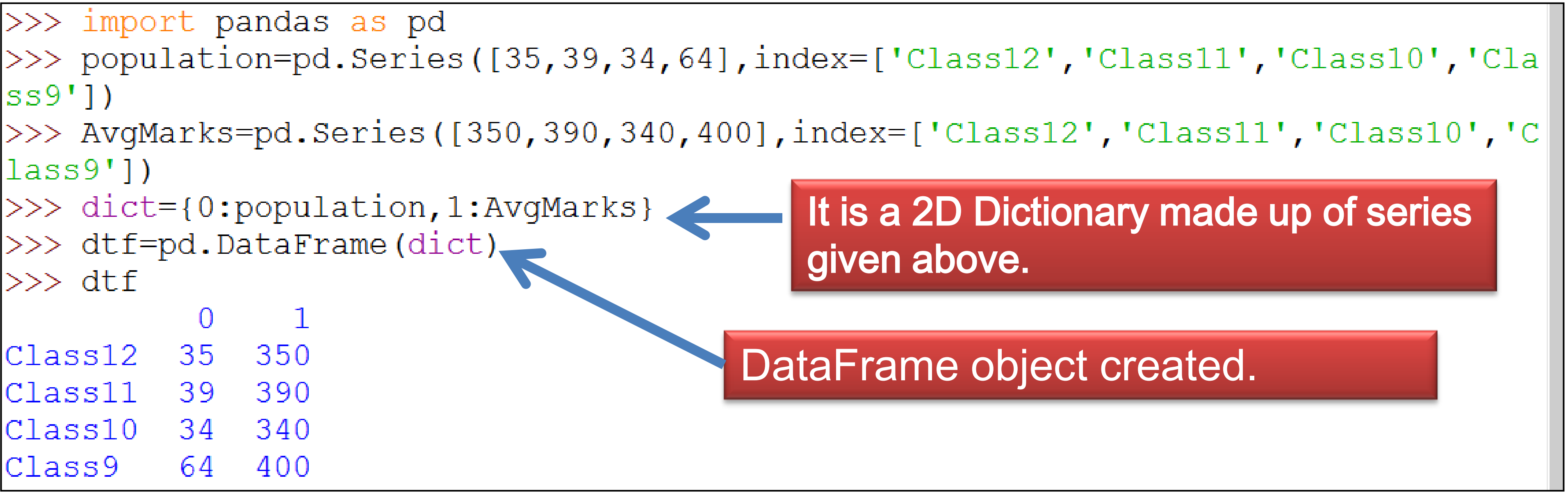
Here, user has given column name .



```
>>> 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 given by user.

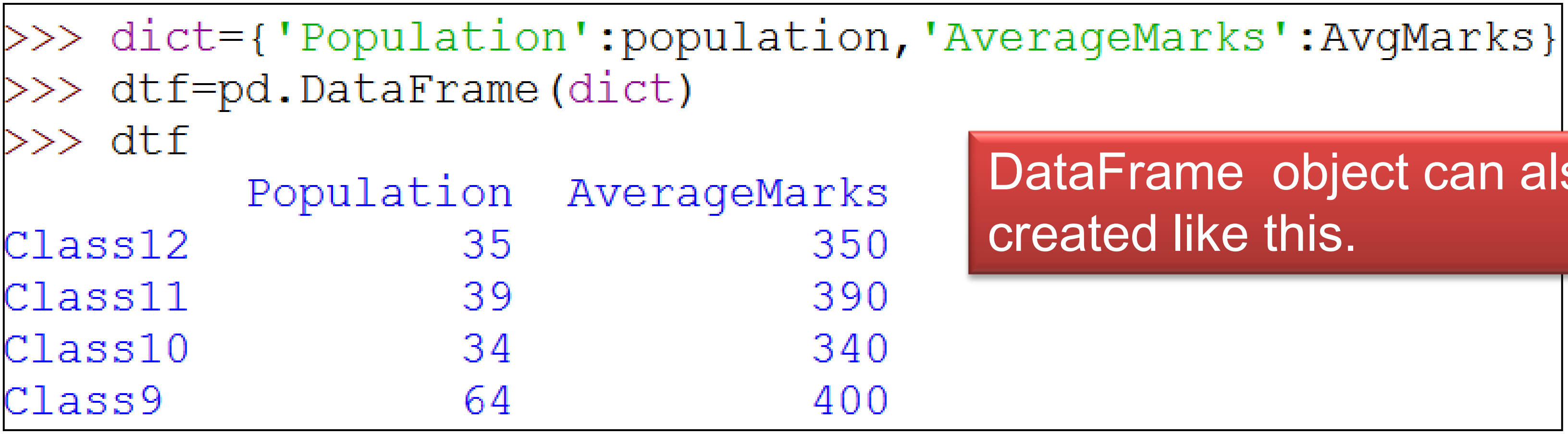
Creation of DataFarme from 2D Dictionary of same Series Object



```
>>> import pandas as pd
>>> population=pd.Series([35,39,34,64],index=['Class12','Class11','Class10','Cla
ss9'])
>>> AvgMarks=pd.Series([350,390,340,400],index=['Class12','Class11','Class10','C
lass9'])
>>> dict={0:population,1:AvgMarks}
>>> dtf=pd.DataFrame(dict)
>>> dtf
          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.

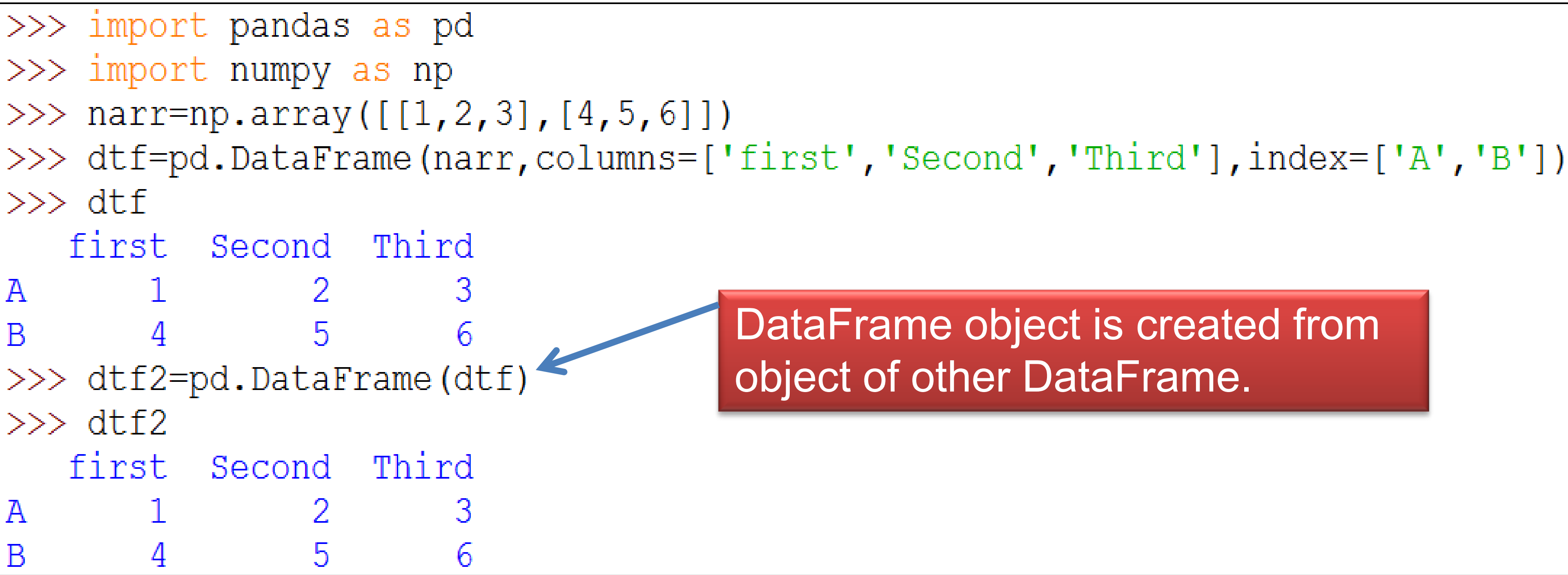
DataFrame object created.



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

DataFrame object can also be created like this.

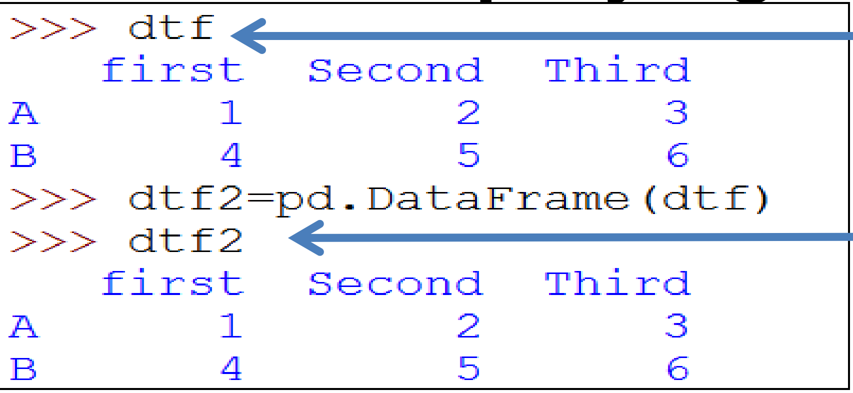
Creation of DataFrame from object of other DataFrame



```
>>> 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
```

DataFrame object is created from object of other DataFrame.

Displaying DataFrame Object



```
>>> 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
```

Syntax for displaying DataFrame object.

DataFrame Attributes

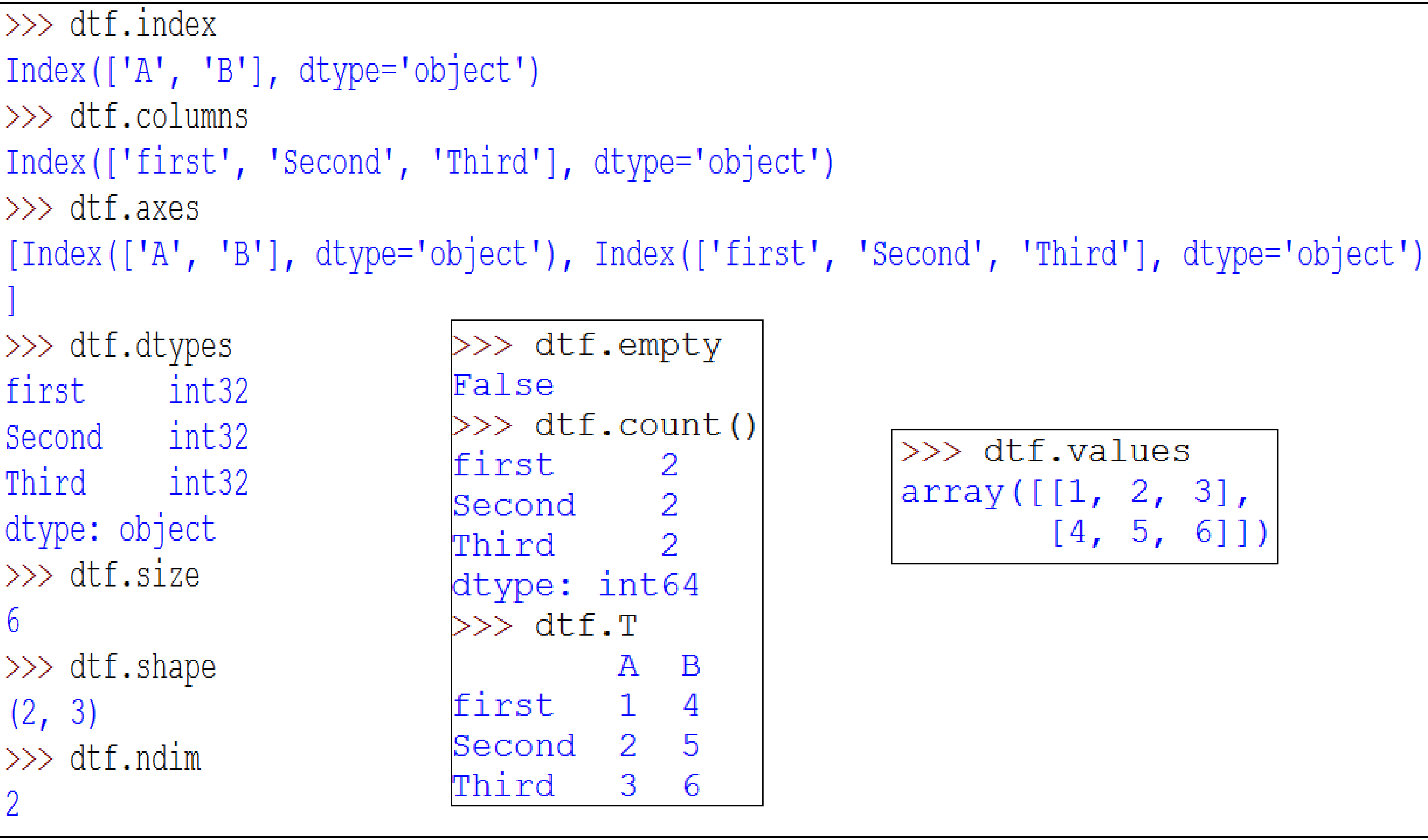
- When we create an object of a DataFrame then all information related to it like size, datatype etc can be accessed by attributes.

  <DataFrame Object>.<attribute name>

- Some attributes are -

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

DataFrame Attributes



```
>>> 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.size
6
>>> dtf.shape
(2, 3)
>>> dtf.ndim
2
```

```
>>> dtf.empty
False
>>> dtf.count()
first     2
Second    2
Third     2
dtype: int64
>>> dtf.T
        A  B
first   1  4
Second  2  5
Third   3  6
```

```
>>> dtf.values
array([[1, 2, 3],
       [4, 5, 6]])
```

Selecting and Accessing from DataFrame

- Selecting a Column-

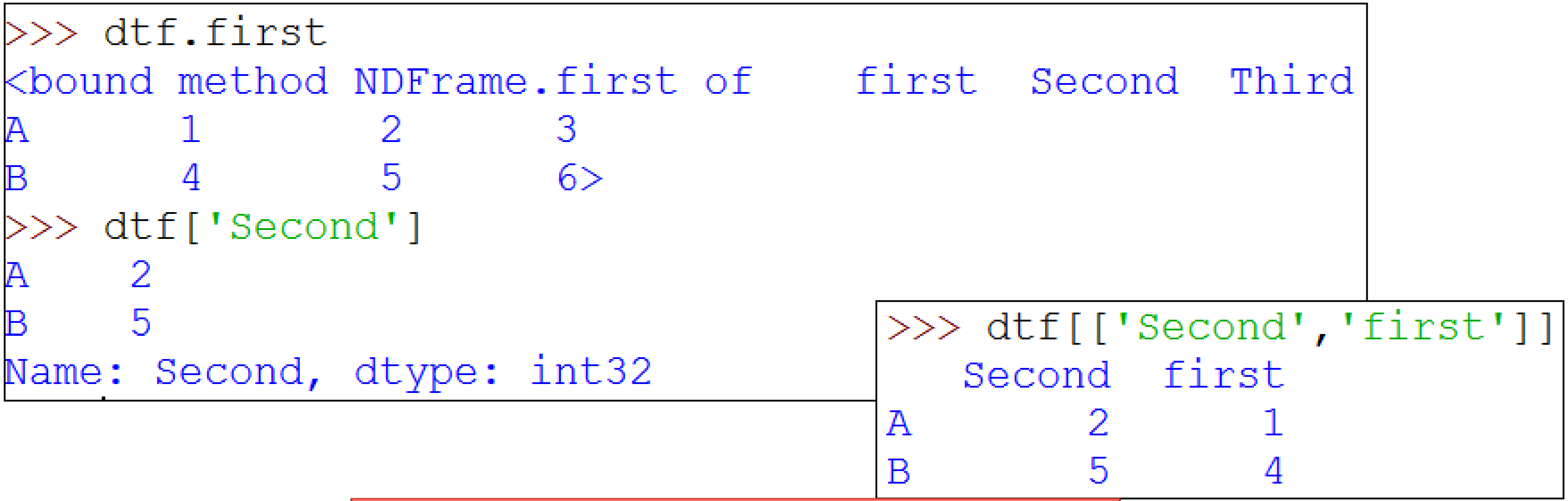
<DataFrame Object>[<column name>]

To select a column

or <DataFrame Object>.<column name>

Selection of multiple column

<DataFrame Object>[List of column name]



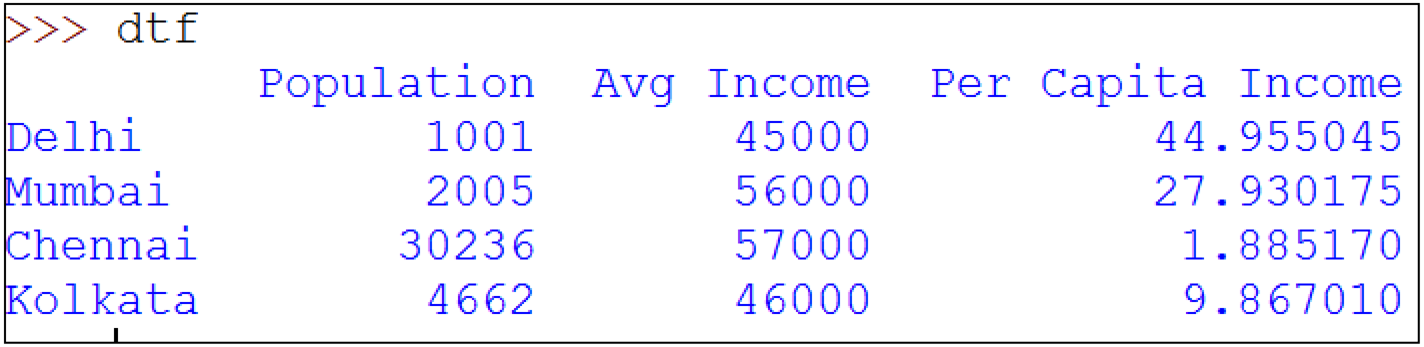
```
>>> dtf.first
<bound method NDFrame.first of     first  Second  Third
A      1       2      3
B      4       5      6>
>>> dtf['Second']
A    2
B    5
Name: Second, dtype: int32
```

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

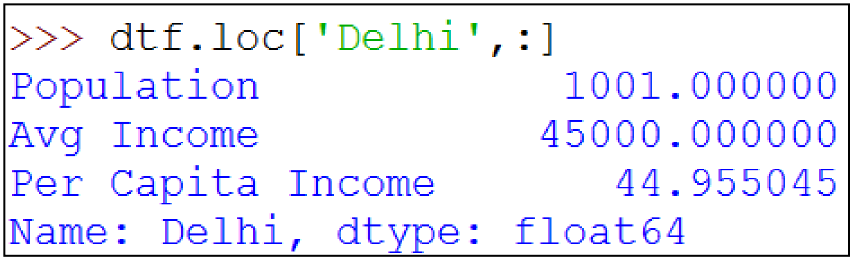
We can change the order in column.

Selection of subset from DataFrame

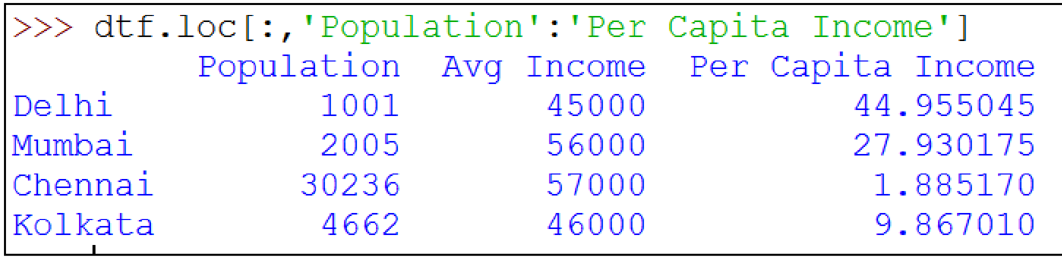
<DataFrameObject>.loc [<StartRow> : <EndRow>, <StartCol> : <EndCol>]



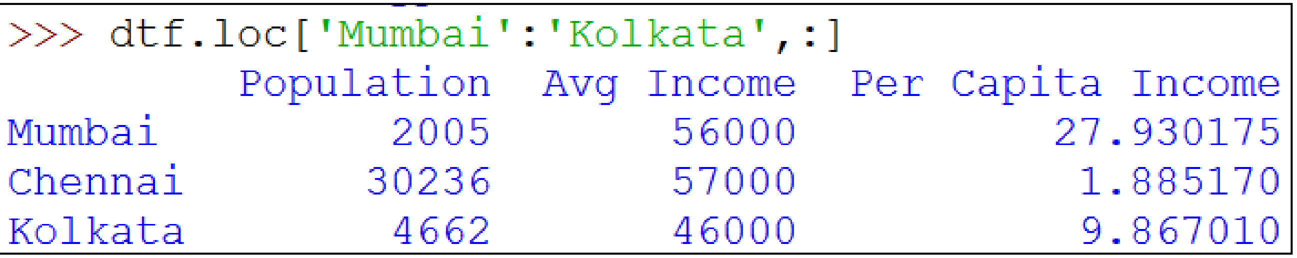
```
>>> dtf
         Population  Avg Income  Per Capita Income
Delhi          1001       45000          44.955045
Mumbai         2005       56000          27.930175
Chennai       30236       57000           1.885170
Kolkata        4662       46000           9.867010
```



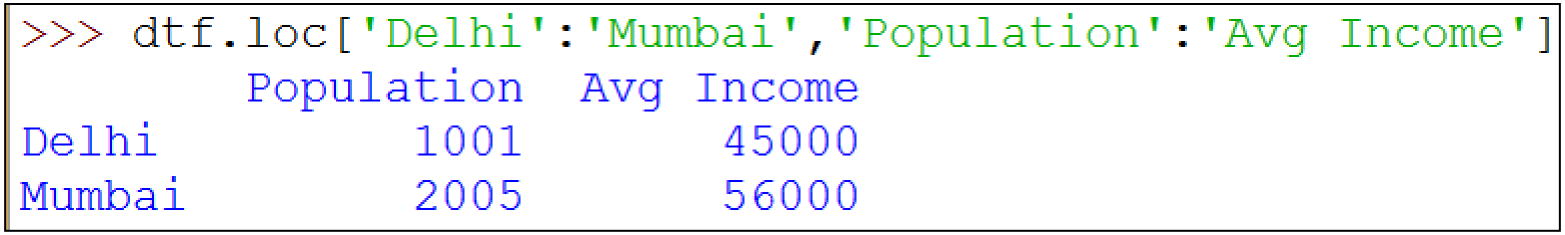
```
>>> dtf.loc['Delhi',:]
Population            1001.000000
Avg Income           45000.000000
Per Capita Income       44.955045
Name: Delhi, dtype: float64
```



```
>>> dtf.loc[:,'Population':'Per Capita Income']
         Population  Avg Income  Per Capita Income
Delhi          1001       45000          44.955045
Mumbai         2005       56000          27.930175
Chennai       30236       57000           1.885170
Kolkata        4662       46000           9.867010
```



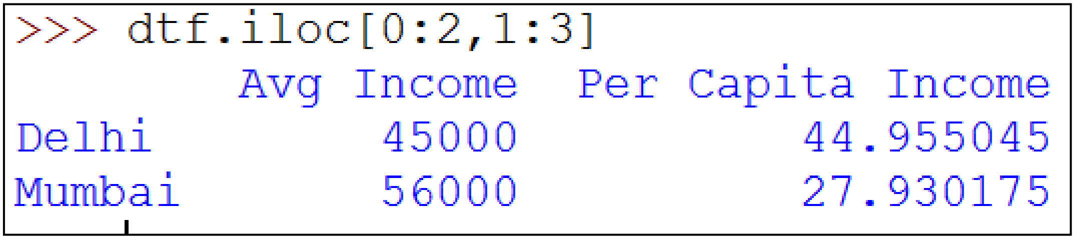
```
>>> dtf.loc['Mumbai':'Kolkata',:]
         Population  Avg Income  Per Capita Income
Mumbai         2005       56000          27.930175
Chennai       30236       57000           1.885170
Kolkata        4662       46000           9.867010
```



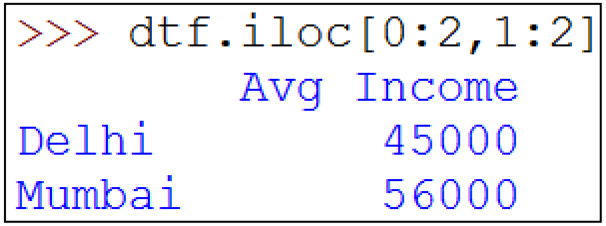
```
>>> dtf.loc['Delhi':'Mumbai','Population':'Avg Income']
        Population  Avg Income
Delhi         1001       45000
Mumbai        2005       56000
```

Selection of subset from DataFrame

<DataFrameObject> .iloc [<Row Index> : <RowIndex>, <ColIndex> : <ColIndex>]



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



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

Selection of an Individual Value from DataFrame

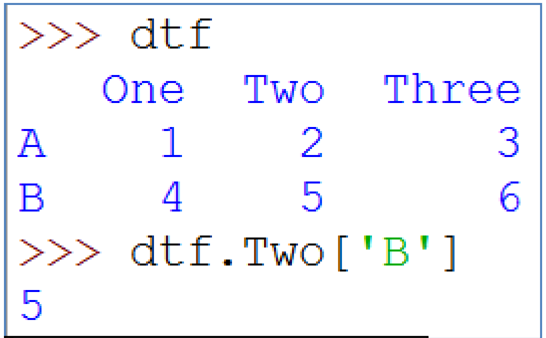
<DFObject>. <col name.[row name or row index]

or

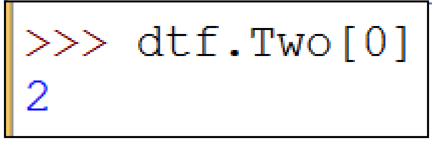
<DFObject> . at [<row name>,<col name>]

or

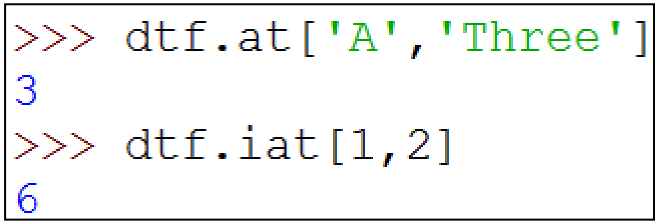
<DFObject> iat[<row index>, <col index>]



```
>>> dtf
   One  Two  Three
A    1    2      3
B    4    5      6
>>> dtf.Two['B']
5
```



```
>>> dtf.Two[0]
2
```

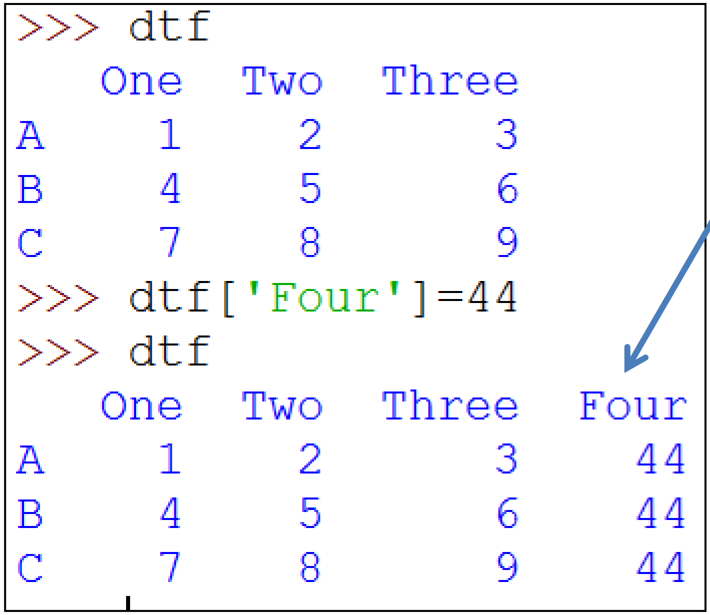


```
>>> dtf.at['A','Three']
3
>>> dtf.iat[1,2]
6
```

Accessing and modifying values in DataFrame

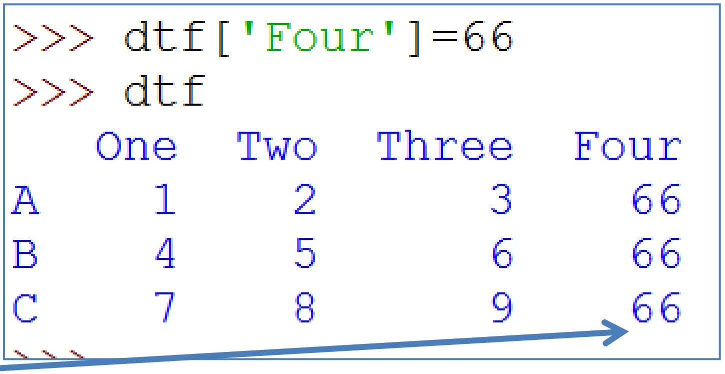
a) Syntax to add or change a column-

<DFObject>.<Col Name>[<row label>]=<new value>



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



```
>>> dtf['Four']=66
>>> dtf
   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’.

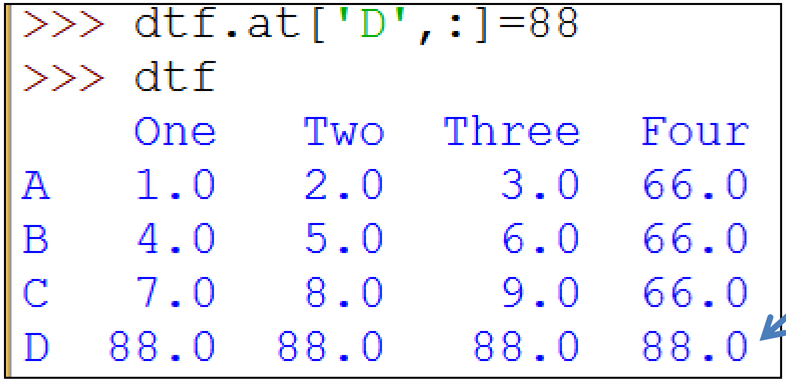
Accessing and modifying values in DataFrame

b) Syntax to add or change a row-

<DFObject> at[<RowName>, :] =<new value>

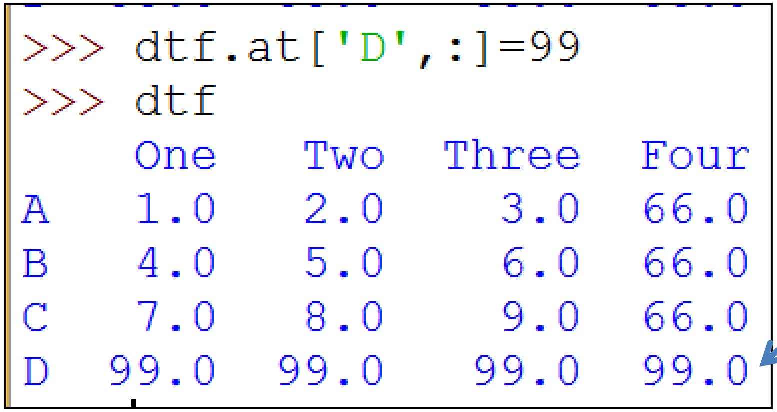
या

<DFObject> loc[<RowName>, :] =<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
```

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



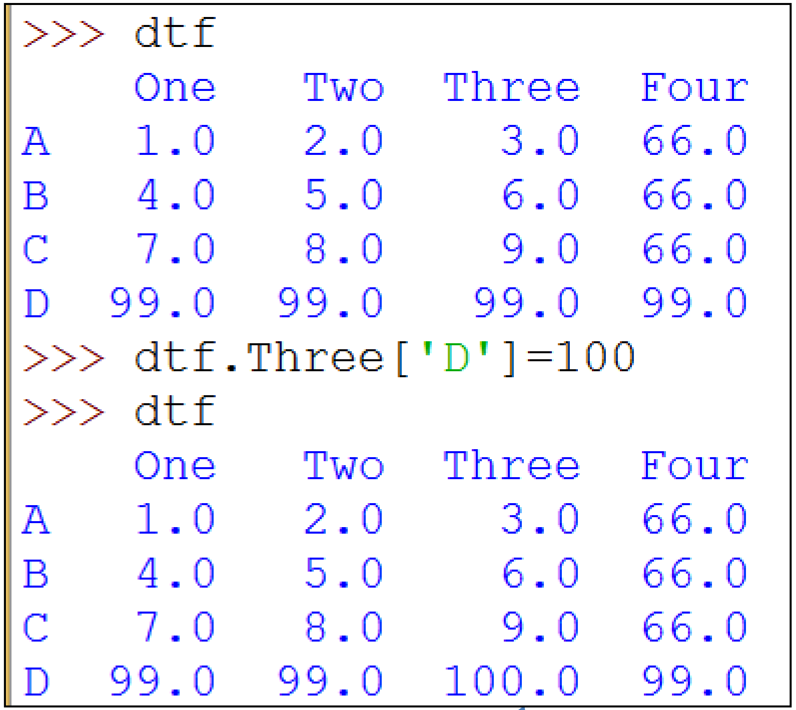
```
>>> dtf.at['D',:]=99
>>> dtf
     One   Two  Three  Four
A    1.0   2.0    3.0  66.0
B    4.0   5.0    6.0  66.0
C    7.0   8.0    9.0  66.0
D   99.0  99.0   99.0  99.0
```

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

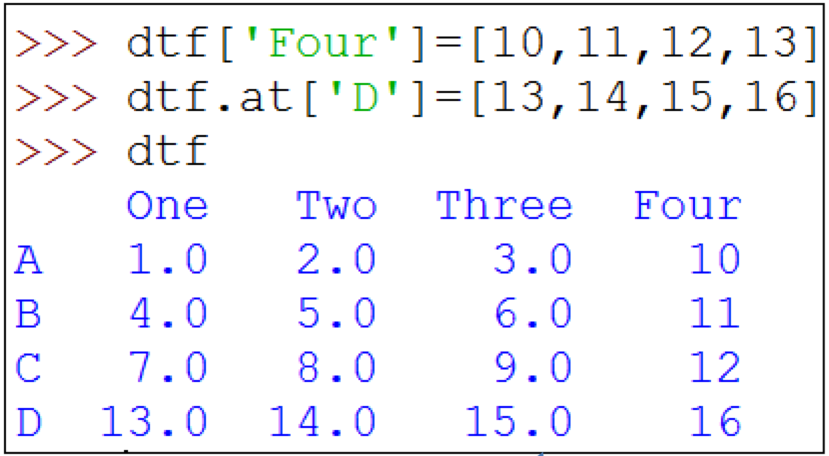
Accessing and modifying values in DataFrame

c) Syntax to change single value-

<DFObject>.<ColName>[<RowName/Lebel>]



```
>>> 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
>>> dtf.Three['D']=100
>>> 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  100.0  99.0
```



```
>>> dtf['Four']=[10,11,12,13]
>>> dtf.at['D']=[13,14,15,16]
>>> dtf
    One   Two  Three  Four
A   1.0   2.0    3.0    10
B   4.0   5.0    6.0    11
C   7.0   8.0    9.0    12
D  13.0  14.0   15.0    16
```

Values can be changed like this also. Values of row and column can be given separately.

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

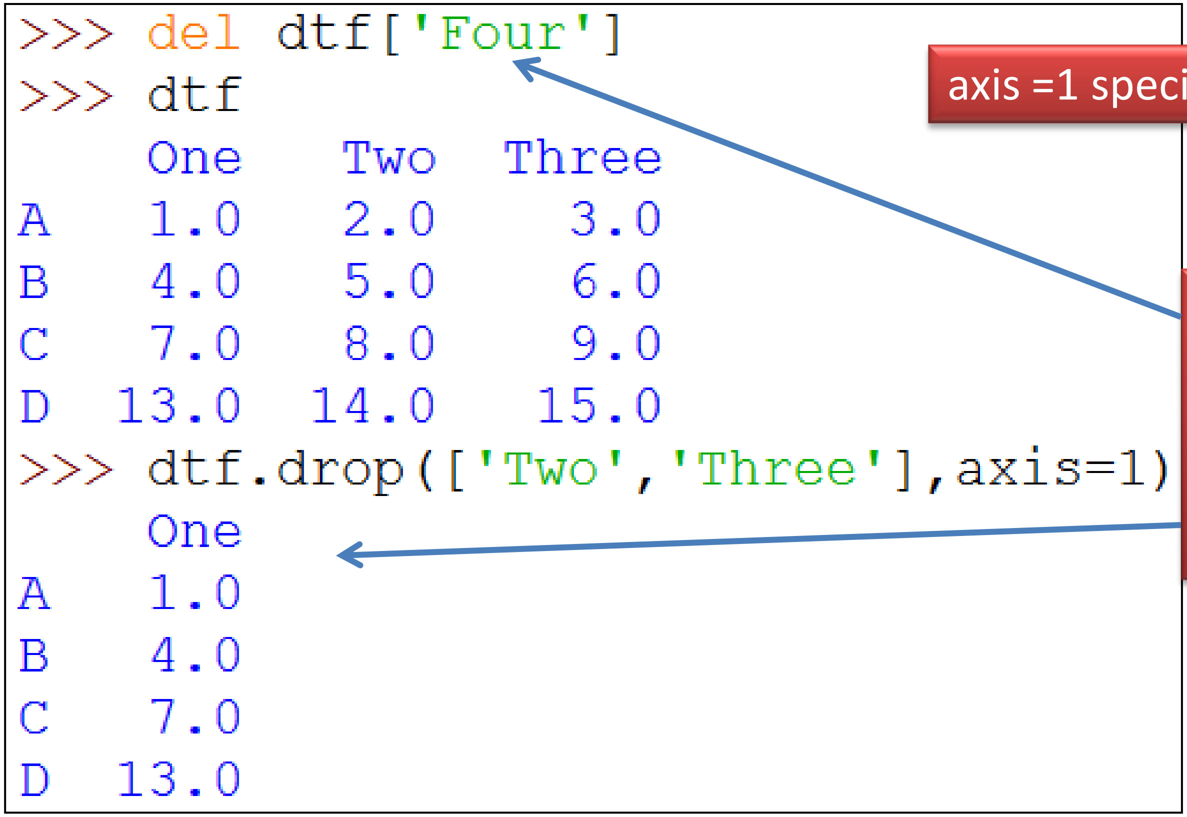
Accessing and modifying values in DataFrame

d) Syntax for Column deletion-

del <DFObject>[<ColName>] or

df.drop([<Col1Name>,<Col2Name>, . .], axis=1)

axis =1 specifies deletion of column.



```
>>> del dtf['Four']
>>> dtf
    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
>>> dtf.drop(['Two','Three'],axis=1)
    One
A   1.0
B   4.0
C   7.0
D  13.0
```

del command does not return value after deletion whereas drop method returns the value to dataframe after deletion.

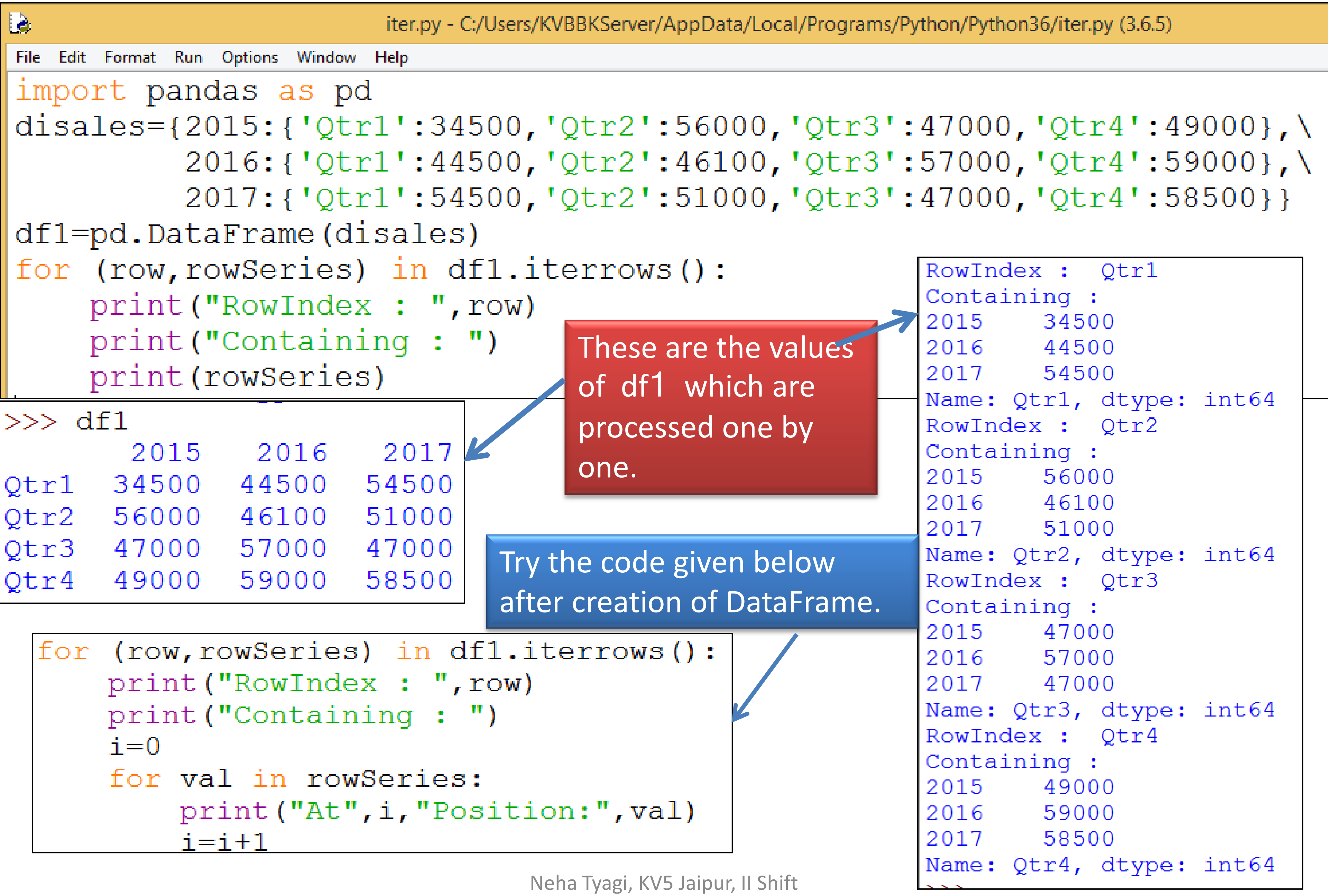
Iteration in DataFrame

- 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-

- <DFObject>.iterrows( ) it represents dataframe in row-wise subsets .

- <DFObject>.iteritems( ) it represents dataframe in column-wise subsets.

Use of pandas.iterrows () function



```
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
```

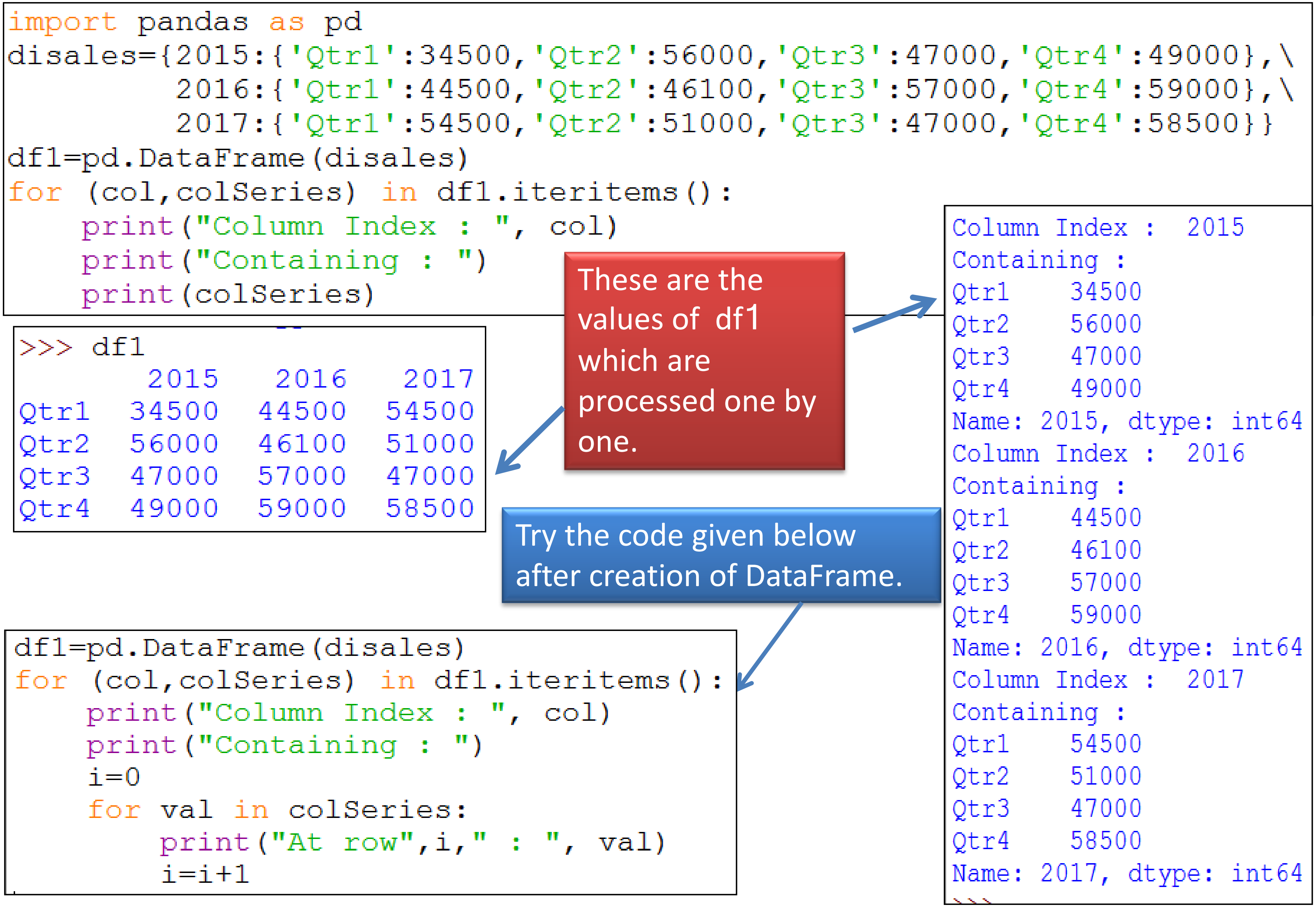
These are the values of df1 which are processed one by one.

```
RowIndex :  Qtr1
Containing :
2015    34500
2016    44500
2017    54500
Name: Qtr1, dtype: int64
RowIndex :  Qtr2
Containing :
2015    56000
2016    46100
2017    51000
Name: Qtr2, dtype: int64
RowIndex :  Qtr3
Containing :
2015    47000
2016    57000
2017    47000
Name: Qtr3, dtype: int64
RowIndex :  Qtr4
Containing :
2015    49000
2016    59000
2017    58500
Name: Qtr4, dtype: int64
```

Try the code given below after creation of DataFrame.

```
for (row,rowSeries) in df1.iterrows():
    print("RowIndex : ",row)
    print("Containing : ")
    i=0
    for val in rowSeries:
        print("At",i,"Position:",val)
        i=i+1
```

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.iteritems():
    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.

```
Column Index :  2015
Containing :
Qtr1     34500
Qtr2     56000
Qtr3     47000
Qtr4     49000
Name: 2015, dtype: int64
Column Index :  2016
Containing :
Qtr1     44500
Qtr2     46100
Qtr3     57000
Qtr4     59000
Name: 2016, dtype: int64
Column Index :  2017
Containing :
Qtr1     54500
Qtr2     51000
Qtr3     47000
Qtr4     58500
Name: 2017, dtype: int64
>>>
```

Try the code given below after creation of DataFrame.

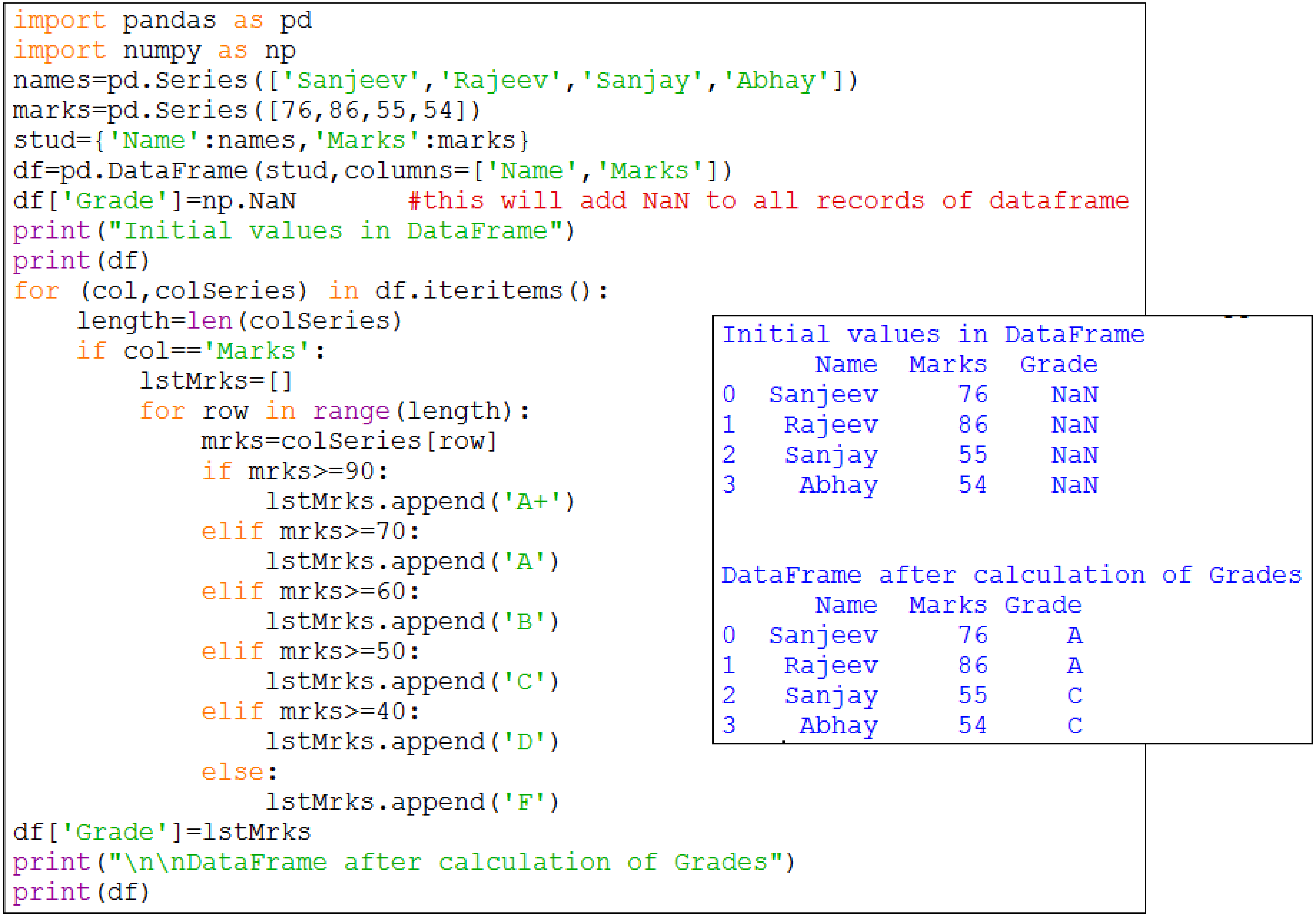
```python
df1=pd.DataFrame(disales)
for (col,colSeries) in df1.iteritems():
    print("Column Index : ", col)
    print("Containing : ")
    i=0
    for val in colSeries:
        print("At row",i," : ", val)
        i=i+1
```

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 > =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

Program for iteration



```
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("\n\nDataFrame after calculation of Grades")
print(df)
```

```
Initial values in DataFrame
      Name  Marks  Grade
0  Sanjeev     76    NaN
1   Rajeev     86    NaN
2   Sanjay     55    NaN
3    Abhay     54    NaN


DataFrame after calculation of Grades
      Name  Marks Grade
0  Sanjeev     76     A
1   Rajeev     86     A
2   Sanjay     55     C
3    Abhay     54     C
```

Binary Operations in a DataFrame

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

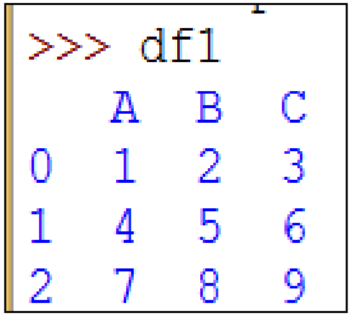
To Add - (+, add or radd)

To Subtract - (-, sub or rsub)

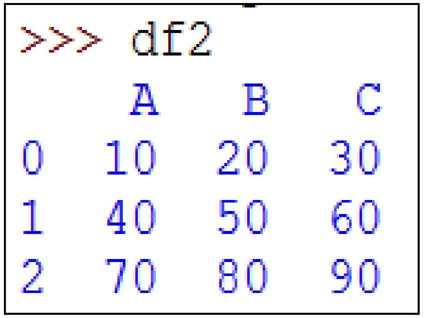
To Multiply– (* or mul)

To Divide - (/ or div)

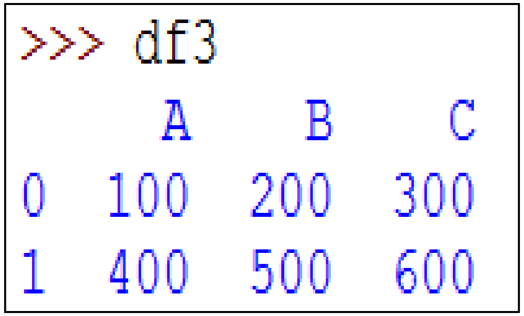
We will perform operations on following dataframes-



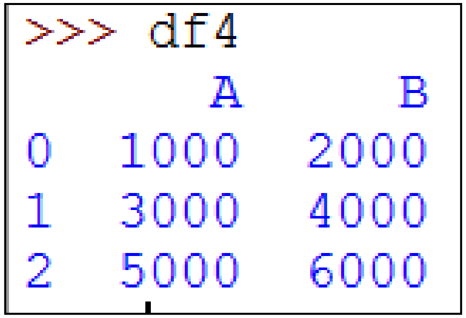
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



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

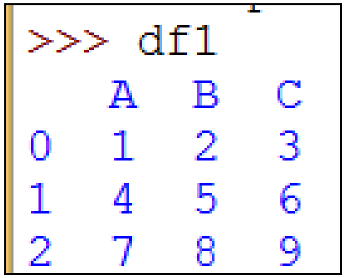


```
>>> df3
     A    B    C
0  100  200  300
1  400  500  600
```

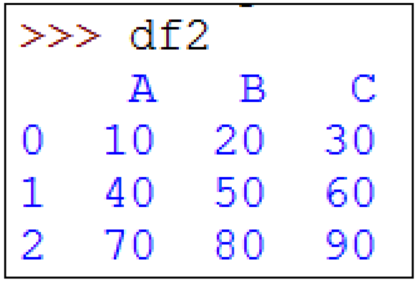


```
>>> df4
      A     B
0  1000  2000
1  3000  4000
2  5000  6000
```

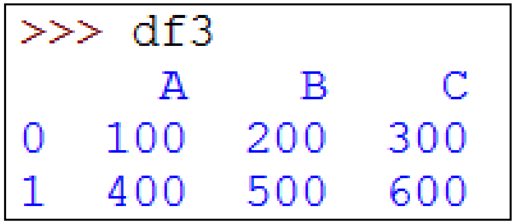
Addition



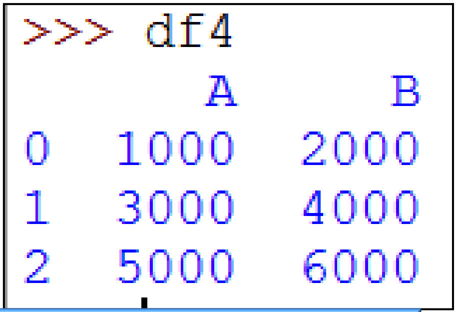
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



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



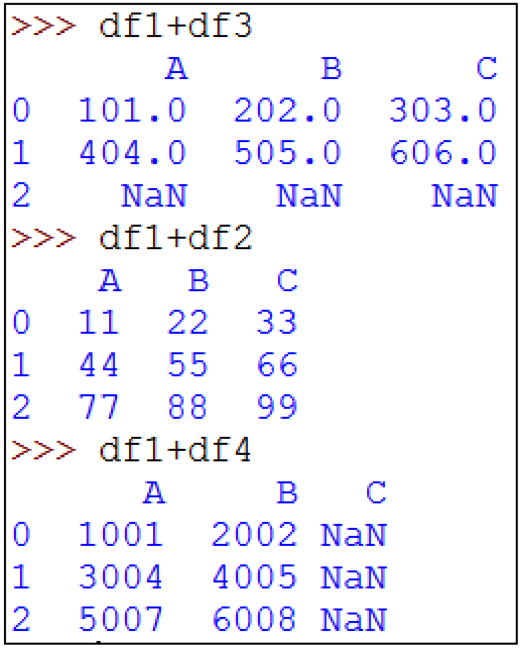
```
>>> df3
     A    B    C
0  100  200  300
1  400  500  600
```



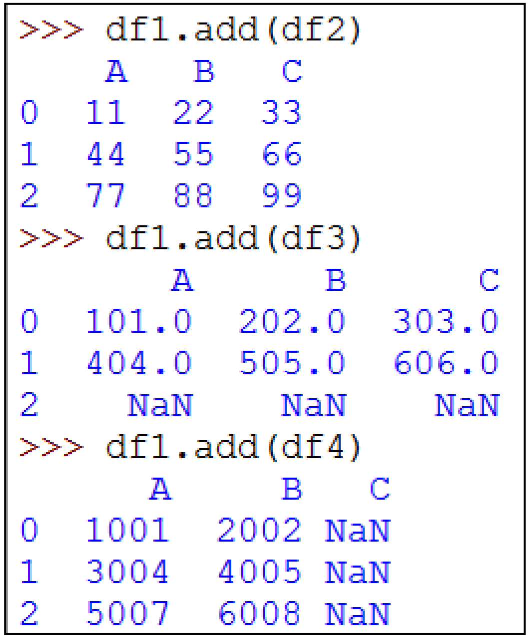
```
>>> df4
      A     B
0  1000  2000
1  3000  4000
2  5000  6000
```

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.

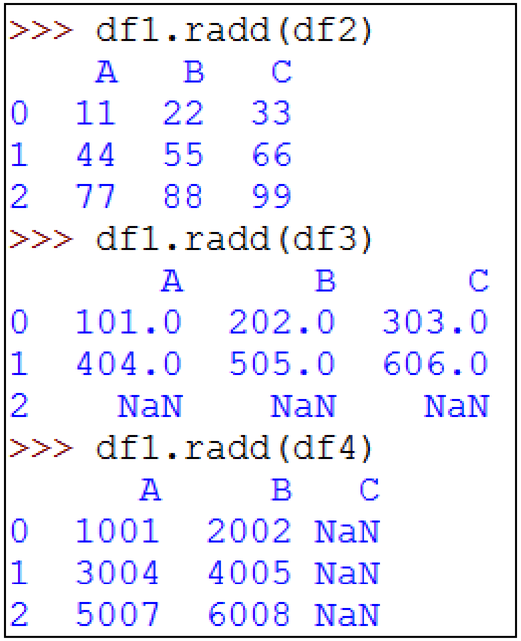
This behavior of ‘data alignment’ on the basis of “matching indexes” is called MATCHING.



```
>>> df1+df3
       A      B      C
0  101.0  202.0  303.0
1  404.0  505.0  606.0
2    NaN    NaN    NaN
>>> df1+df2
    A   B   C
0  11  22  33
1  44  55  66
2  77  88  99
>>> df1+df4
      A     B   C
0  1001  2002 NaN
1  3004  4005 NaN
2  5007  6008 NaN
```

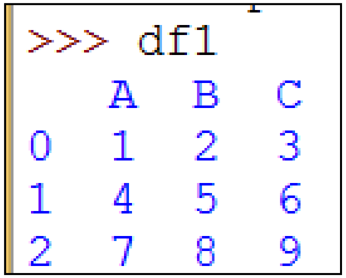


```
>>> df1.add(df2)
    A   B   C
0  11  22  33
1  44  55  66
2  77  88  99
>>> df1.add(df3)
       A      B      C
0  101.0  202.0  303.0
1  404.0  505.0  606.0
2    NaN    NaN    NaN
>>> df1.add(df4)
      A     B   C
0  1001  2002 NaN
1  3004  4005 NaN
2  5007  6008 NaN
```

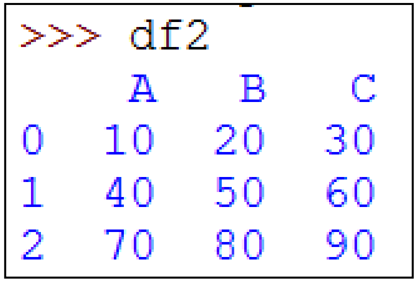


```
>>> df1.radd(df2)
    A   B   C
0  11  22  33
1  44  55  66
2  77  88  99
>>> df1.radd(df3)
       A      B      C
0  101.0  202.0  303.0
1  404.0  505.0  606.0
2    NaN    NaN    NaN
>>> df1.radd(df4)
      A     B   C
0  1001  2002 NaN
1  3004  4005 NaN
2  5007  6008 NaN
```

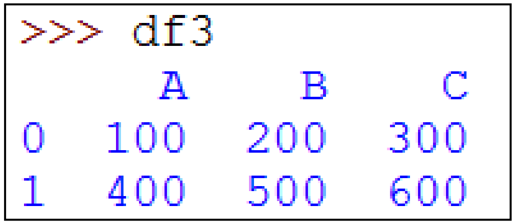
Subtraction



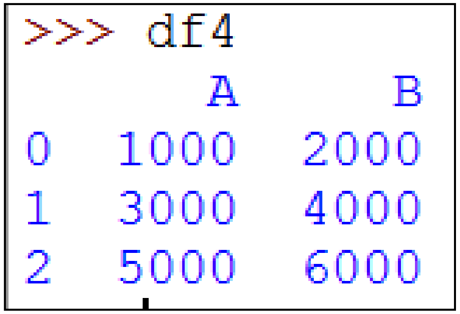
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



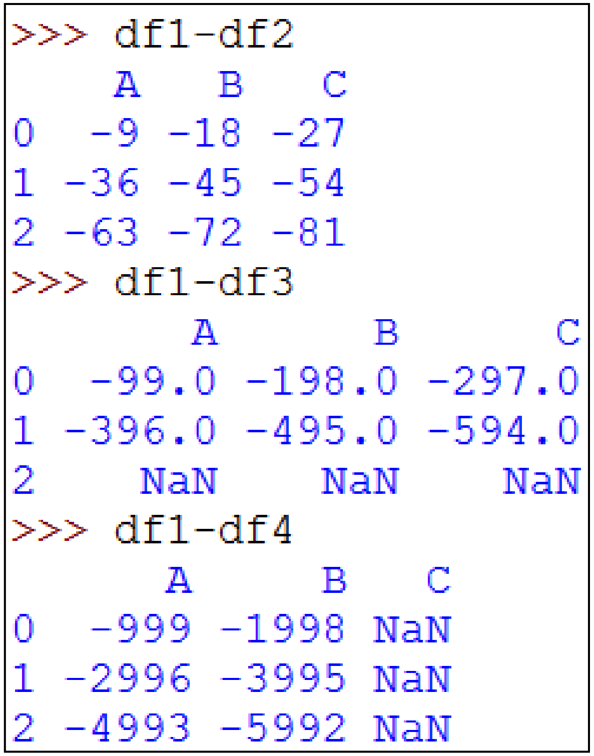
```
>>> df2
    A   B   C
0  10  20  30
1  40  50  60
2  70  80  90
```



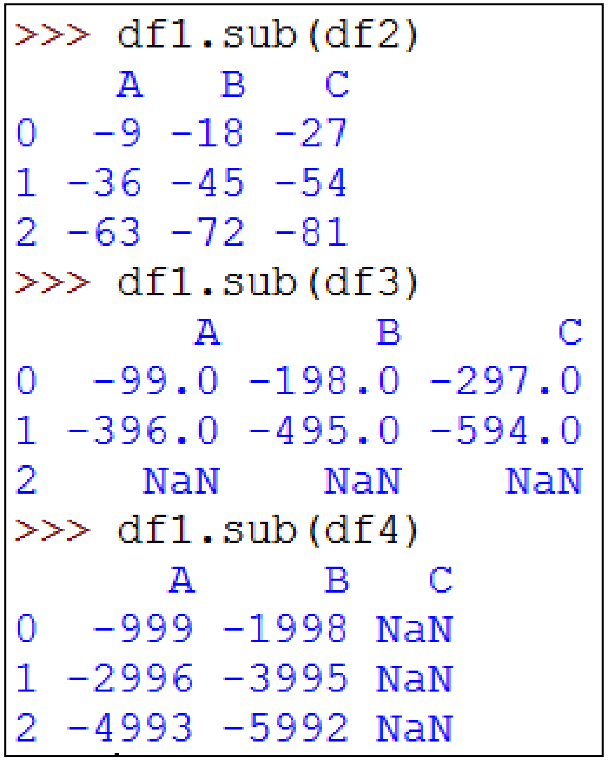
```
>>> df3
     A    B    C
0  100  200  300
1  400  500  600
```



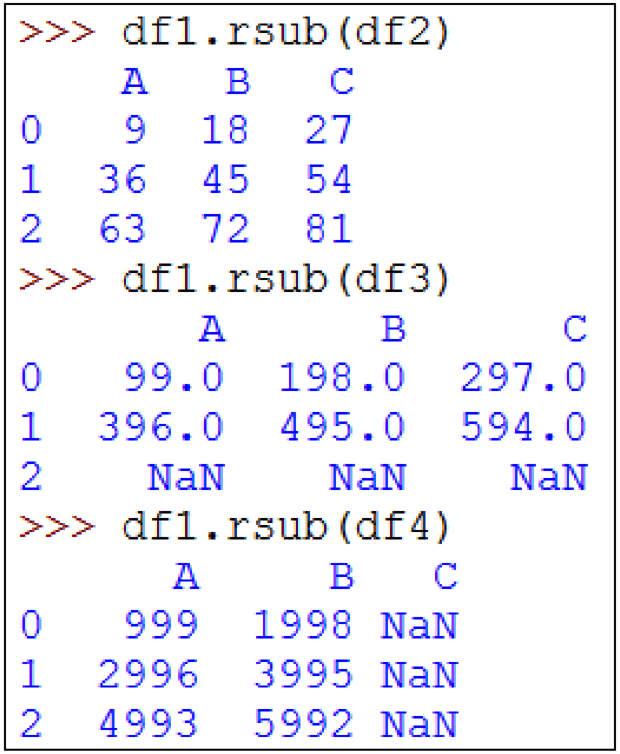
```
>>> df4
      A     B
0  1000  2000
1  3000  4000
2  5000  6000
```



```
>>> 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
```

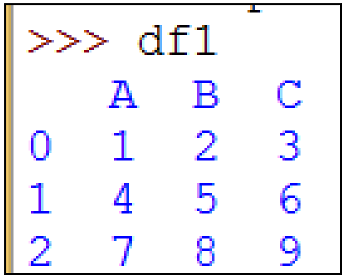


```
>>> df1.sub(df2)
    A   B   C
0  -9 -18 -27
1 -36 -45 -54
2 -63 -72 -81
>>> df1.sub(df3)
       A      B      C
0  -99.0 -198.0 -297.0
1 -396.0 -495.0 -594.0
2    NaN    NaN    NaN
>>> df1.sub(df4)
      A     B   C
0  -999 -1998 NaN
1 -2996 -3995 NaN
2 -4993 -5992 NaN
```

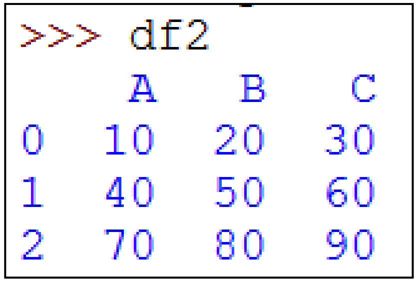


```
>>> df1.rsub(df2)
    A   B   C
0   9  18  27
1  36  45  54
2  63  72  81
>>> df1.rsub(df3)
       A      B      C
0   99.0  198.0  297.0
1  396.0  495.0  594.0
2    NaN    NaN    NaN
>>> df1.rsub(df4)
      A     B   C
0   999  1998 NaN
1  2996  3995 NaN
2  4993  5992 NaN
```

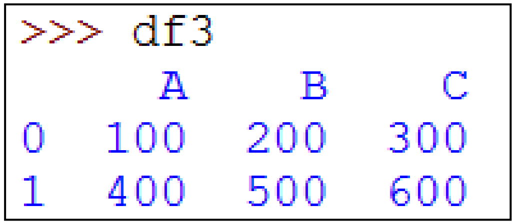
Multiplication



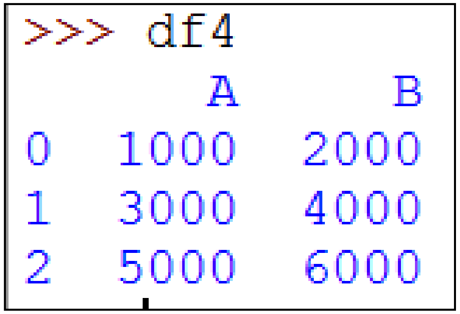
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



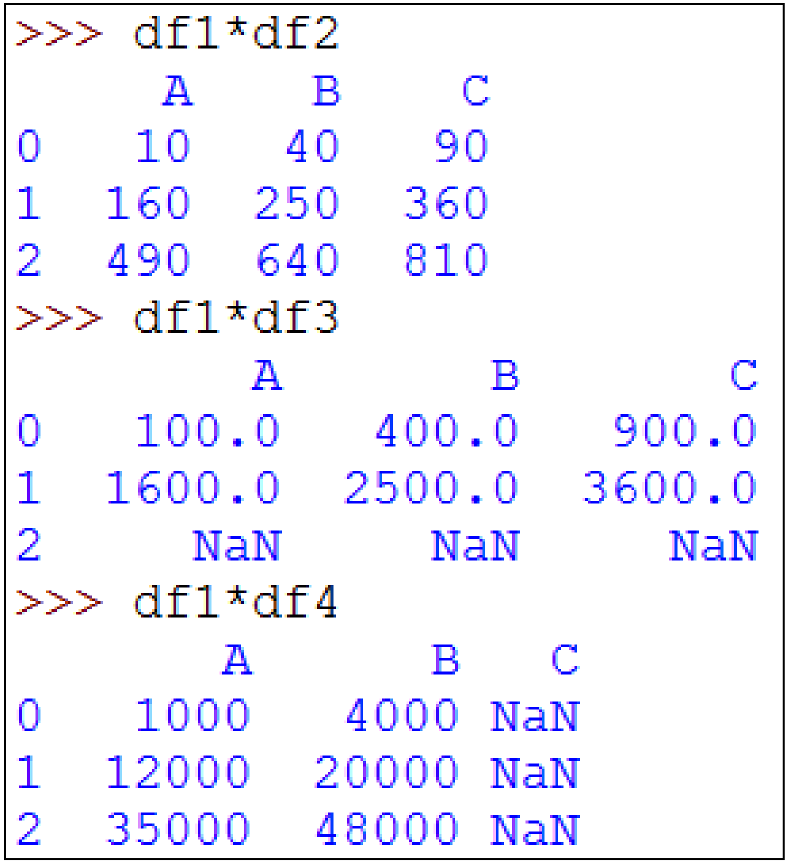
```
>>> df2
    A   B   C
0  10  20  30
1  40  50  60
2  70  80  90
```



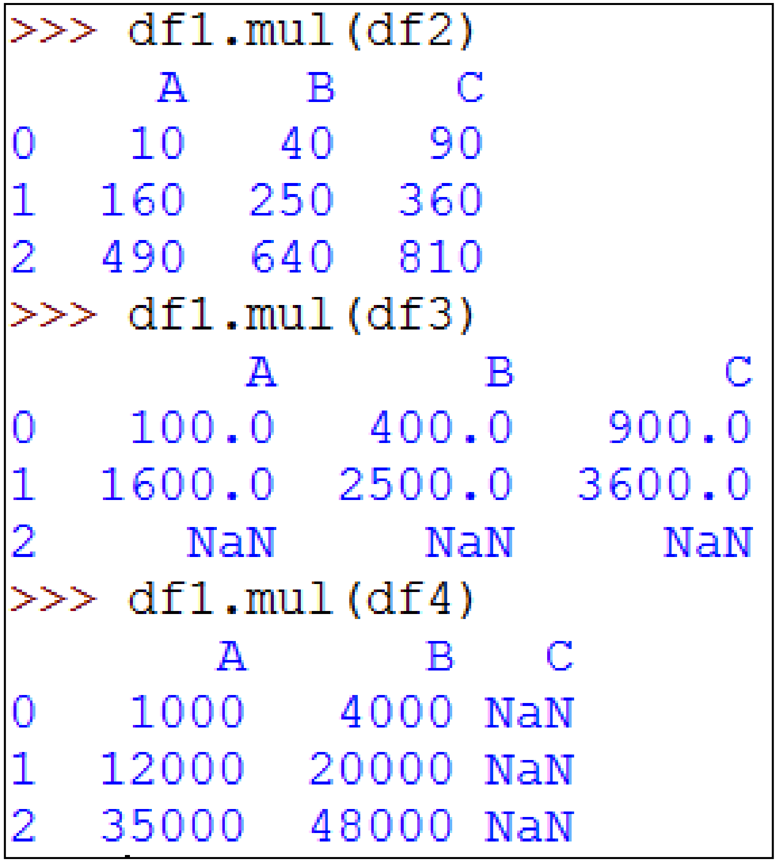
```
>>> df3
     A    B    C
0  100  200  300
1  400  500  600
```



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

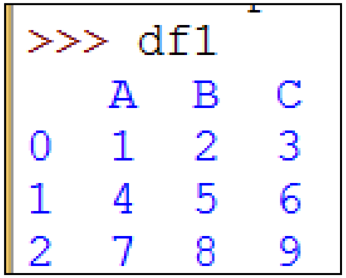


```
>>> df1*df2
     A    B    C
0   10   40   90
1  160  250  360
2  490  640  810
>>> df1*df3
        A       B       C
0   100.0   400.0   900.0
1  1600.0  2500.0  3600.0
2     NaN     NaN     NaN
>>> df1*df4
       A      B   C
0   1000   4000 NaN
1  12000  20000 NaN
2  35000  48000 NaN
```

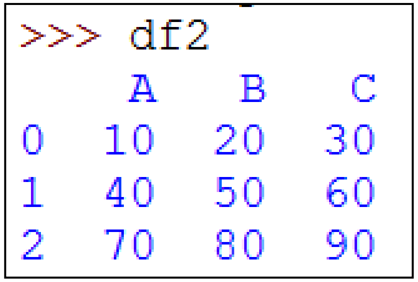


```
>>> df1.mul(df2)
     A    B    C
0   10   40   90
1  160  250  360
2  490  640  810
>>> df1.mul(df3)
        A       B       C
0   100.0   400.0   900.0
1  1600.0  2500.0  3600.0
2     NaN     NaN     NaN
>>> df1.mul(df4)
       A      B   C
0   1000   4000 NaN
1  12000  20000 NaN
2  35000  48000 NaN
```

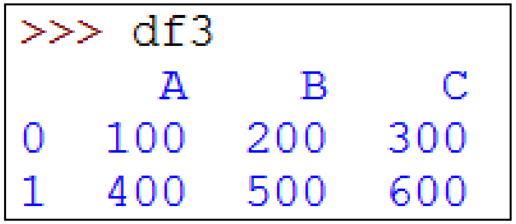
Division



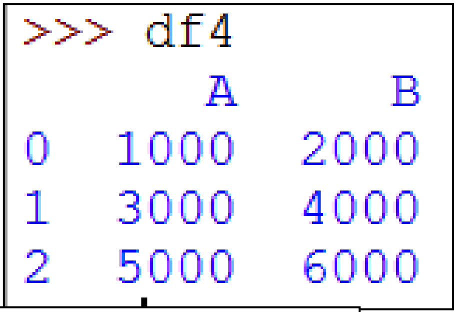
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



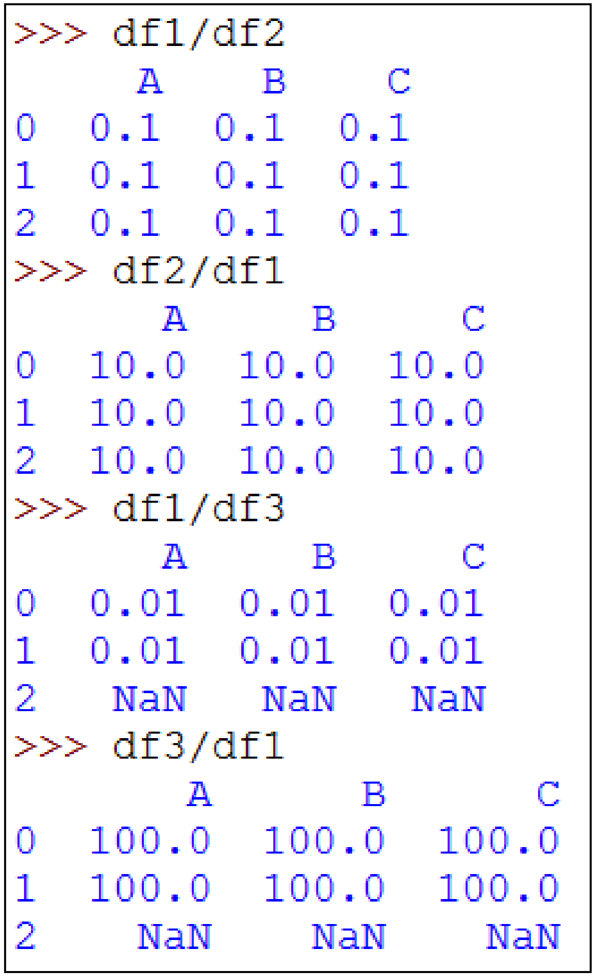
```
>>> df2
    A   B   C
0  10  20  30
1  40  50  60
2  70  80  90
```



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

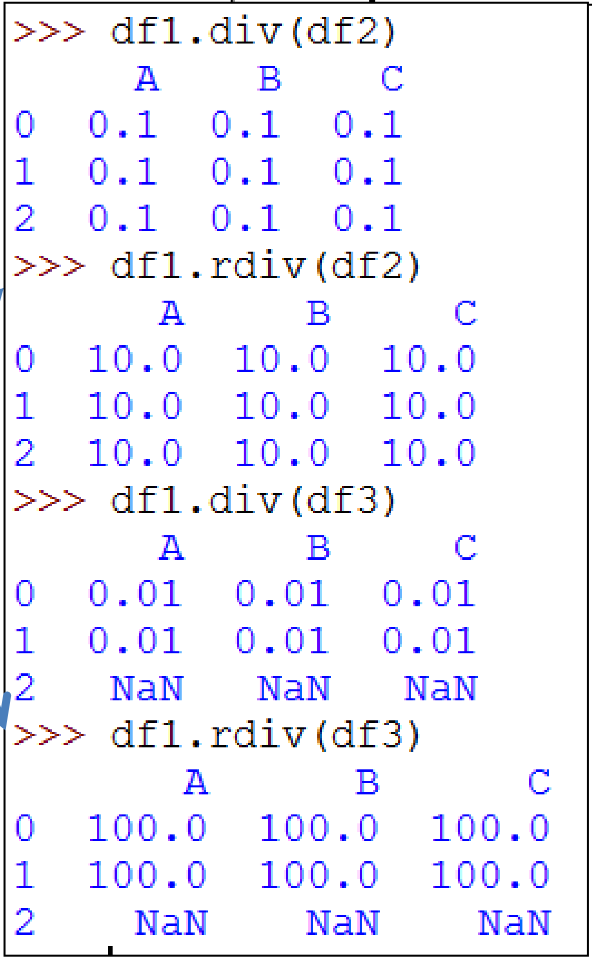


```
>>> df4
      A     B
0  1000  2000
1  3000  4000
2  5000  6000
```



```
>>> df1/df2
     A    B    C
0  0.1  0.1  0.1
1  0.1  0.1  0.1
2  0.1  0.1  0.1
>>> df2/df1
      A     B     C
0  10.0  10.0  10.0
1  10.0  10.0  10.0
2  10.0  10.0  10.0
>>> df1/df3
      A     B     C
0  0.01  0.01  0.01
1  0.01  0.01  0.01
2   NaN   NaN   NaN
>>> df3/df1
       A      B      C
0  100.0  100.0  100.0
1  100.0  100.0  100.0
2    NaN    NaN    NaN
```

See the operation of the rdiv carefully



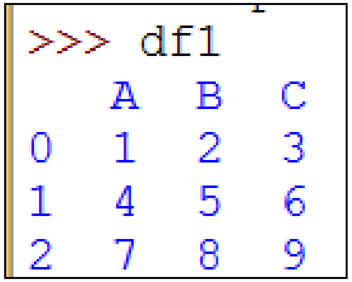
```
>>> df1.div(df2)
     A    B    C
0  0.1  0.1  0.1
1  0.1  0.1  0.1
2  0.1  0.1  0.1
>>> df1.rdiv(df2)
      A     B     C
0  10.0  10.0  10.0
1  10.0  10.0  10.0
2  10.0  10.0  10.0
>>> df1.div(df3)
      A     B     C
0  0.01  0.01  0.01
1  0.01  0.01  0.01
2   NaN   NaN   NaN
>>> df1.rdiv(df3)
       A      B      C
0  100.0  100.0  100.0
1  100.0  100.0  100.0
2    NaN    NaN    NaN
```

Other important functions

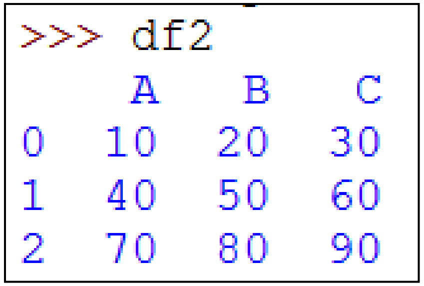
Other important functions of DataFrame are as under-

<DF>.info ( )

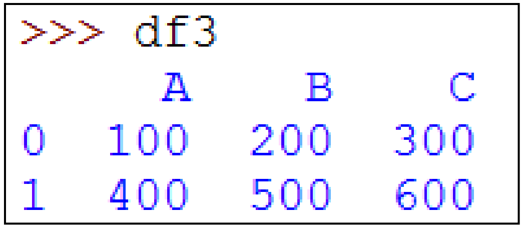
<DF>.describe ( )



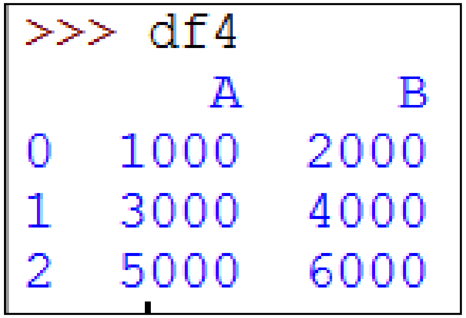
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



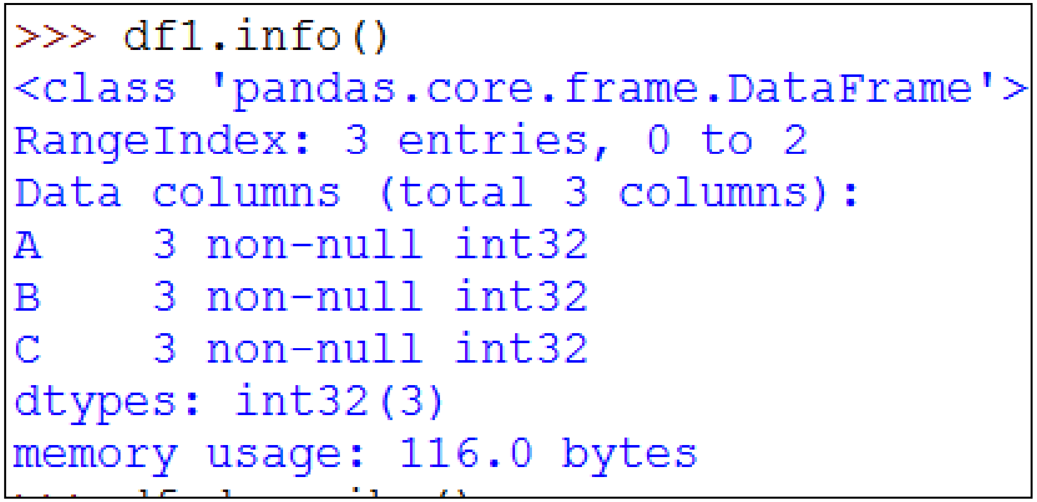
```
>>> df2
    A   B   C
0  10  20  30
1  40  50  60
2  70  80  90
```



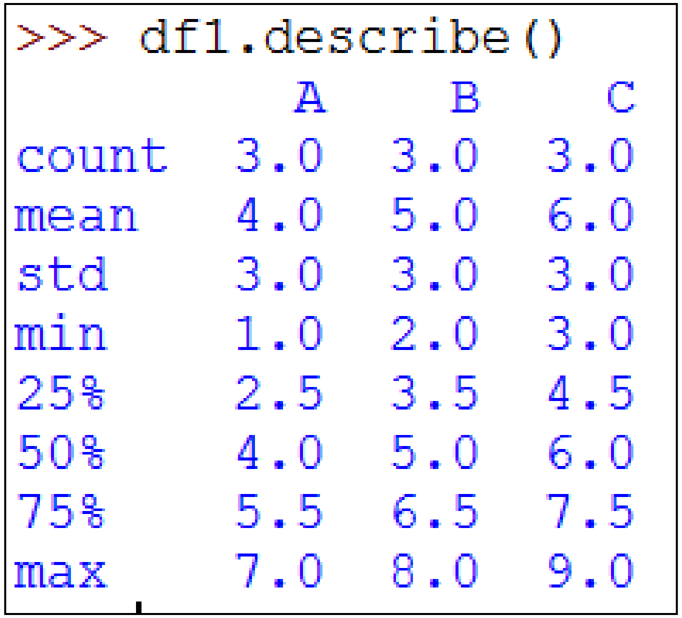
```
>>> df3
     A    B    C
0  100  200  300
1  400  500  600
```



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



```
>>> df1.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
A    3 non-null int32
B    3 non-null int32
C    3 non-null int32
dtypes: int32(3)
memory usage: 116.0 bytes
```



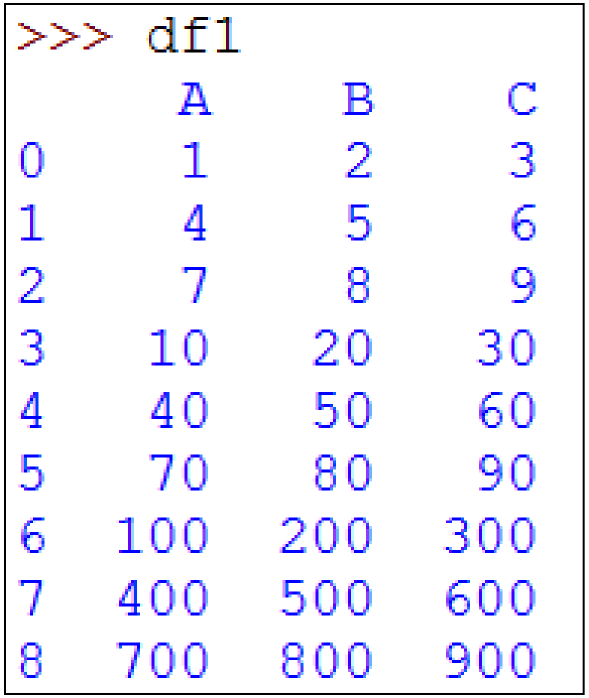
```
>>> df1.describe()
         A    B    C
count  3.0  3.0  3.0
mean   4.0  5.0  6.0
std    3.0  3.0  3.0
min    1.0  2.0  3.0
25%    2.5  3.5  4.5
50%    4.0  5.0  6.0
75%    5.5  6.5  7.5
max    7.0  8.0  9.0
```

Other important functions

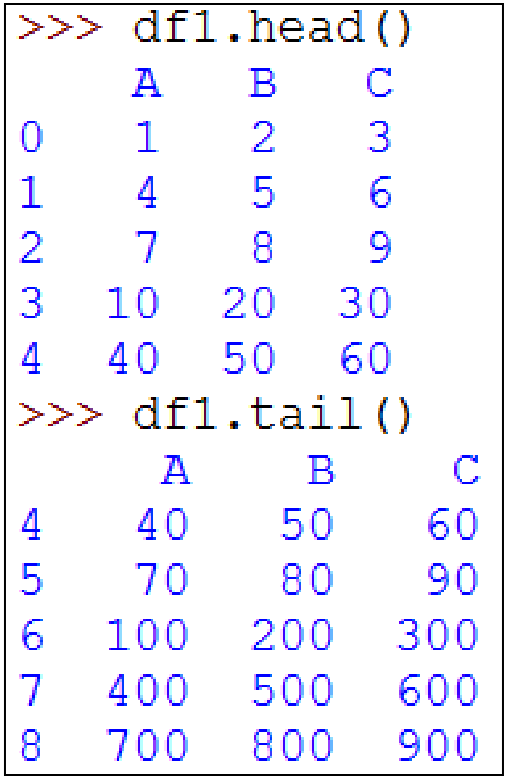
Other important functions of DataFrame are as under-

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

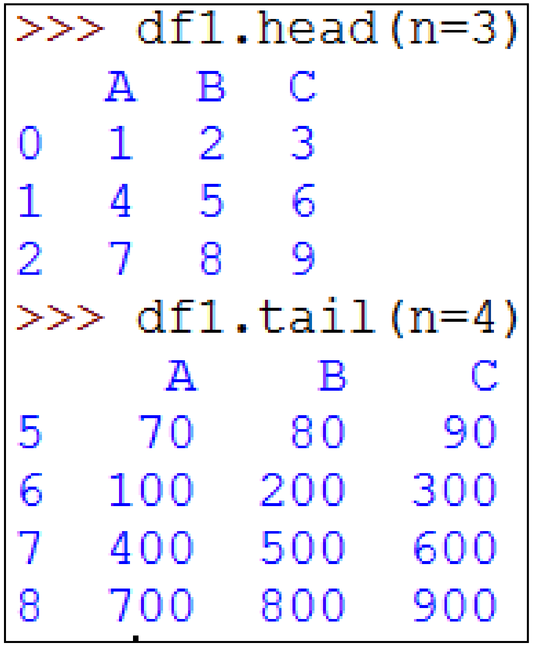
<DF>.tail ([n=<n>])



```
>>> df1
     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
```



```
>>> df1.head()
    A   B   C
0   1   2   3
1   4   5   6
2   7   8   9
3  10  20  30
4  40  50  60
>>> df1.tail()
     A    B    C
4   40   50   60
5   70   80   90
6  100  200  300
7  400  500  600
8  700  800  900
```

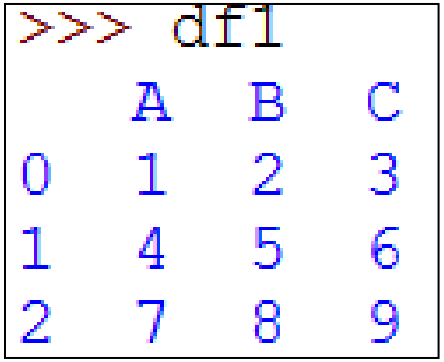


```
>>> df1.head(n=3)
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
>>> df1.tail(n=4)
     A    B    C
5   70   80   90
6  100  200  300
7  400  500  600
8  700  800  900
```

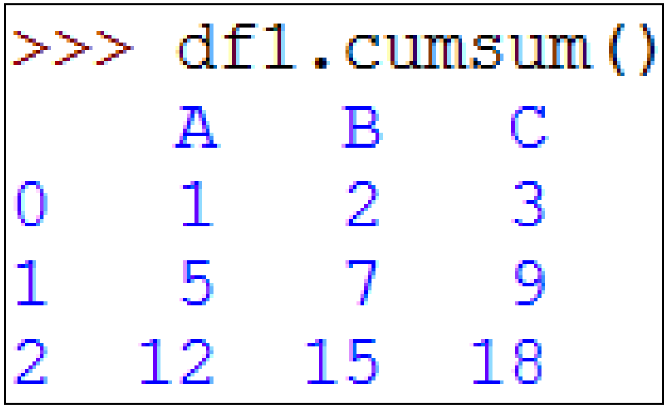
Cumulative Calculations Functions

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

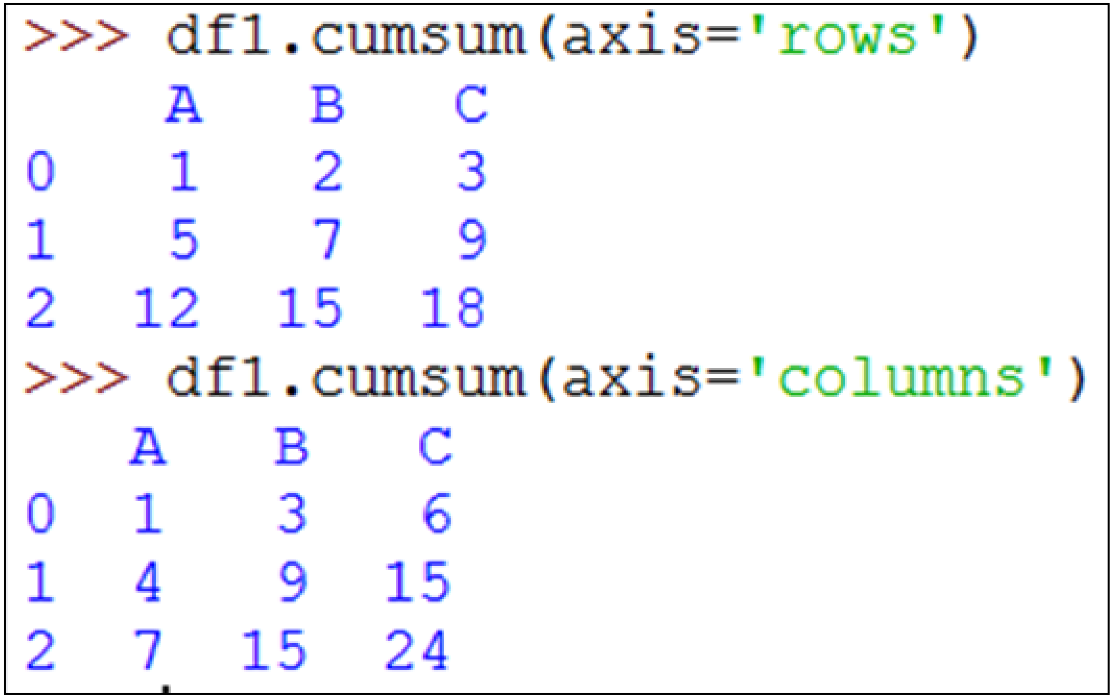
<DF>.cumsum([axis = None]) here, axis argument is optional. |



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

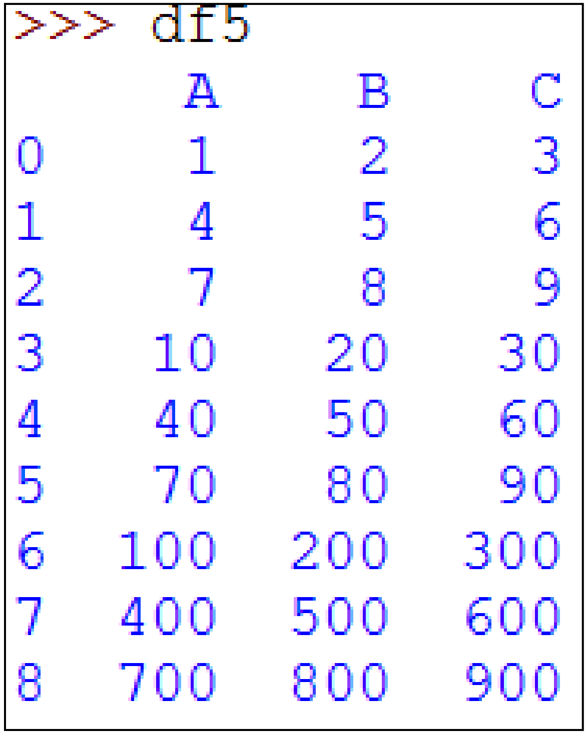


```
>>> df1.cumsum()
    A   B   C
0   1   2   3
1   5   7   9
2  12  15  18
```



```
>>> df1.cumsum(axis='rows')
    A   B   C
0   1   2   3
1   5   7   9
2  12  15  18
>>> df1.cumsum(axis='columns')
   A   B   C
0  1   3   6
1  4   9  15
2  7  15  24
```

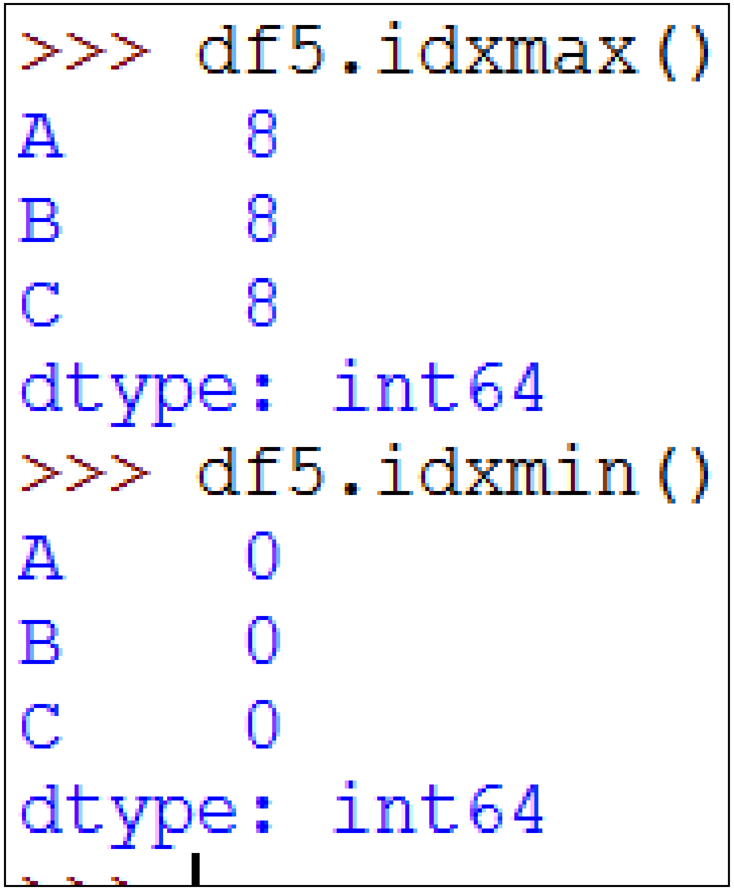
Index of Maximum and Minimum Values



```
>>> 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
```

<DF>.idxmax ( )

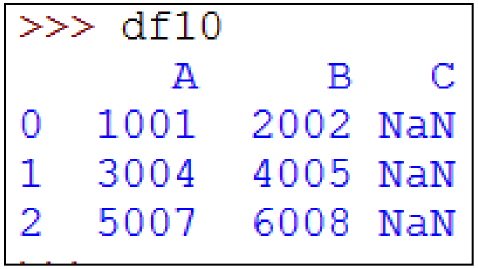
<DF>.idxmin ( )



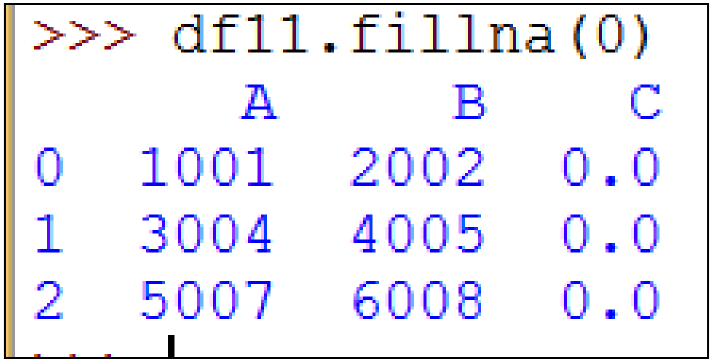
```
>>> df5.idxmax()
A    8
B    8
C    8
dtype: int64
>>> df5.idxmin()
A    0
B    0
C    0
dtype: int64
```

Handling of Missing Data

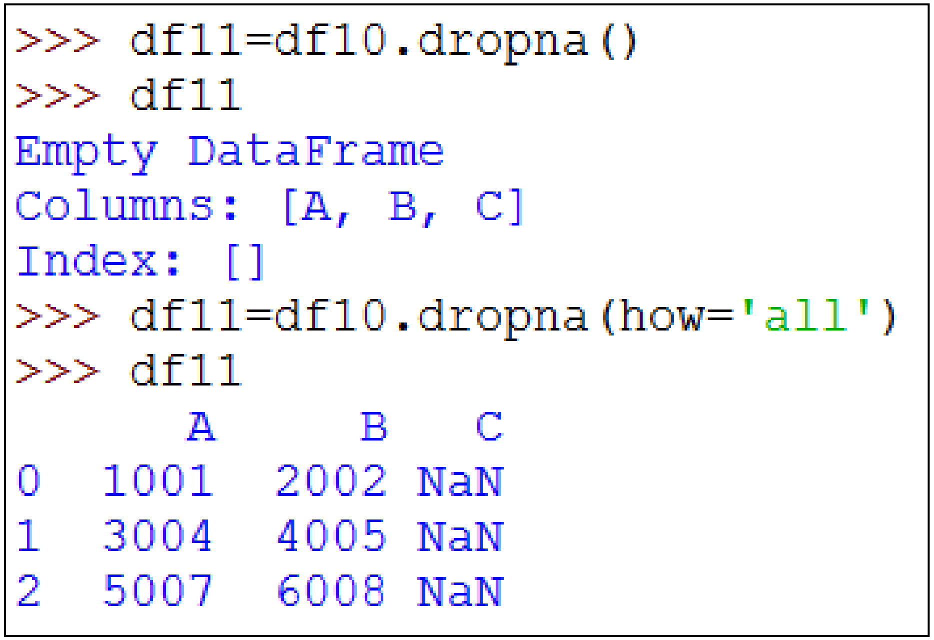
- The values with no computational significance are called missing values.
- Handling methods for missing values-
  - ➢ Dropping missing data
  - ➢ Filling missing data (Imputation)



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

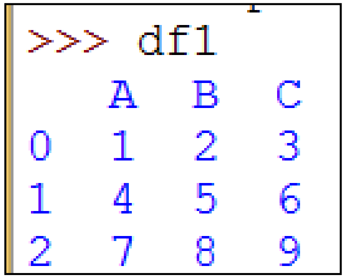


```
>>> df11.fillna(0)
      A     B    C
0  1001  2002  0.0
1  3004  4005  0.0
2  5007  6008  0.0
```

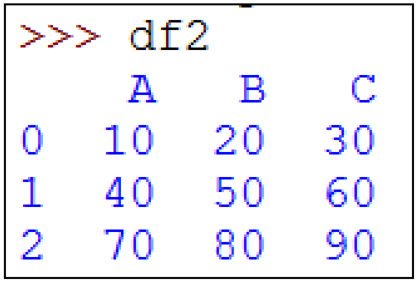


```
>>> df11=df10.dropna()
>>> df11
Empty DataFrame
Columns: [A, B, C]
Index: []
>>> df11=df10.dropna(how='all')
>>> df11
      A     B   C
0  1001  2002 NaN
1  3004  4005 NaN
2  5007  6008 NaN
```

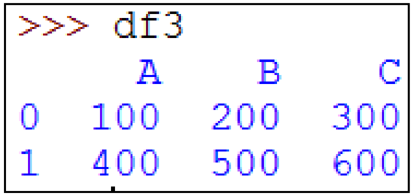
Comparison of Pandas Objects



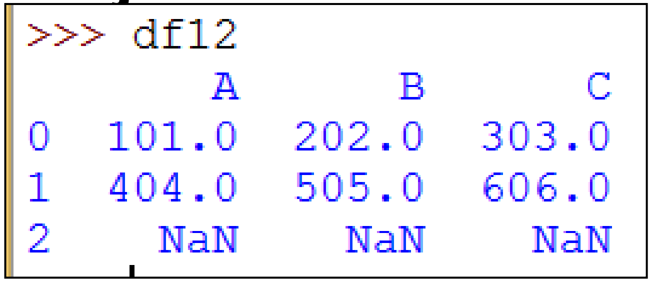
```
>>> df1
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```



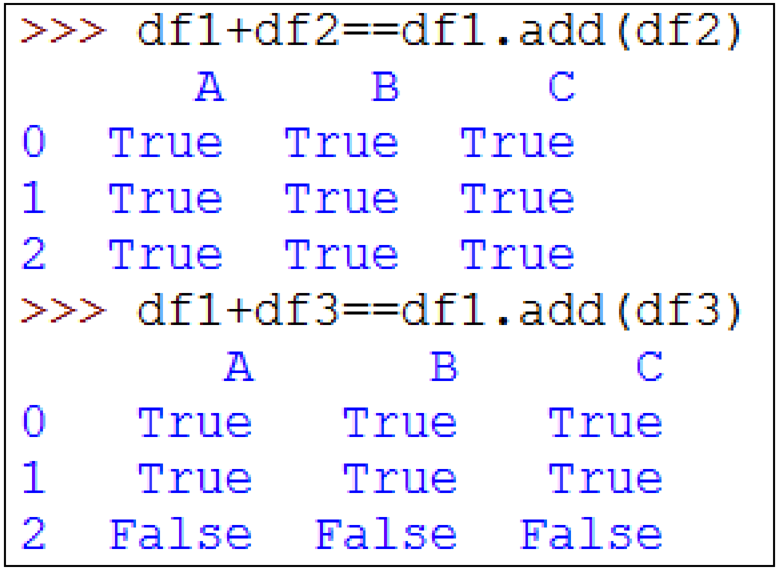
```
>>> df2
    A   B   C
0  10  20  30
1  40  50  60
2  70  80  90
```



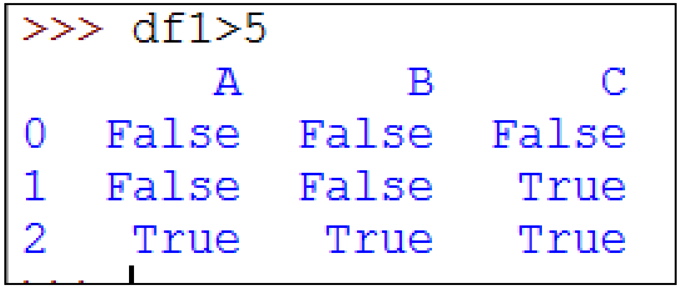
```
>>> df3
     A    B    C
0  100  200  300
1  400  500  600
```



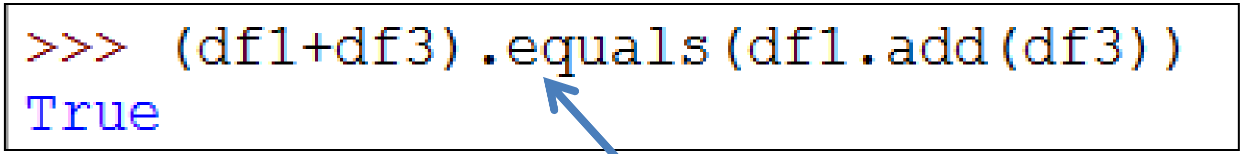
```
>>> df12
       A      B      C
0  101.0  202.0  303.0
1  404.0  505.0  606.0
2    NaN    NaN    NaN
```



```
>>> 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>5
       A      B      C
0  False  False  False
1  False  False   True
2   True   True   True
```



```
>>> (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 | ताकि आपको और सारे chapters मिल सकें |

www.pythontrends.wordpress.com

एक शुरुआत
pythontrends
पाइथन सीखें और सिखाएं

नमस्ते दोस्तों ! /Hello Friends!

यह ब्लॉग उन बच्चों की मदद के लिए बनाया गया है जो python में प्रोग्रामिंग सीख रहे हैं | यह ब्लॉग द्विभाषीय होगा जिससे सीबीएसई बोर्ड के वे बच्चे जिन्हें अंग्रजी भाषा में समस्या होती है उन्हें सही मार्गदर्शन करेगा तथा प्रोग्रामिंग में उनकी सहायता करेगा | जैसा की हम जानते हैं की हमारे देश में कई क्षेत्र और कई लोग ऐसे हैं जिनकी अंग्रेज़ी उतनी मज़बूत नहीं है क्यों कि ये हमारी मातृभाषा नहीं है | तो हमें कभी कभी अंग्रेज़ी के कठिन शब्दों को समझने में समय लगता है और ये समय अगर लॉजिकल विचारों में लगे तो छात्रों का अधिक भला हो सकता है | इस ब्लॉग पर हम कोशिश करेंगे की पाइथन से सम्बंधित सभी तथ्य तथा सामग्री इस ब्लॉग पर उपलब्ध कराएं | यह ब्लॉग **संजीव भदौरिया (पी जी टी कंप्यूटर साइंस) के० वि० बाराबंकी लखनऊ संभाग एवं नेहा त्यागी (पी जी टी कंप्यूटर साइंस) के० वि० क्रं -5 जयपुर,**