5 DataFrame Data Structure

A **DataFrame** in another Pandas structure, which stores data in two-dimensional way. It is actually a **two-dimensional** (tabular and spreadsheet like) **labeled array**, which is actually an **ordered collection of columns** where columns may store different types of data, *e.g., numeric* or *string* or *floating point* or *Boolean* type etc.

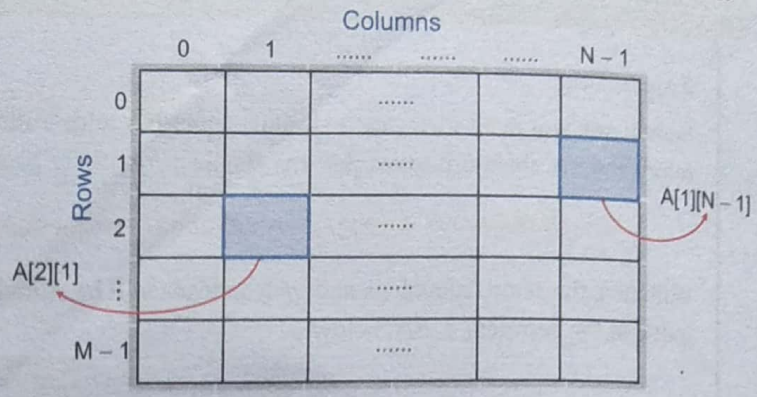
Since DataFrame data structure is like a two-dimensional array, let us first understand what a two-dimensional array is like.

DATAFRAME

> A **DataFrame** is a two-dimensional labeled array like Pandas data structure that stores an ordered collection columns that can store data of different types.

A two-dimensional array is an array in which each element is itself an array. For instance, an array *A* [m] [n] is an *M* by *N* table with *M* rows and *N* columns containing $M \times N$ elements.

The number of elements in a 2-D array can be determined by multiplying number of rows with number of columns. For example, the number of elements in an array A[7] [9] is calculated as $7 \times 9 = 63$.



Characteristics

Major **characteristics of a DataFrame data structure** can be listed as :

(*i*) It has two indexes or we can say that **two axes** – a ***row index*** (***axis = 0***) and a ***column index (axis = 1)***. [Fig. 1.7(*b*)]

(*ii*) Conceptually it is like a spreadsheet where each value is identifiable with the combination of *row index* and *column index*. The ***row index*** is known as index in general and the ***column index*** is called the column-name. [Fig. 1.7(*b*)]

(*iii*) The indexes can be of numbers or letters or strings. [Fig. 1.7(*a*)]

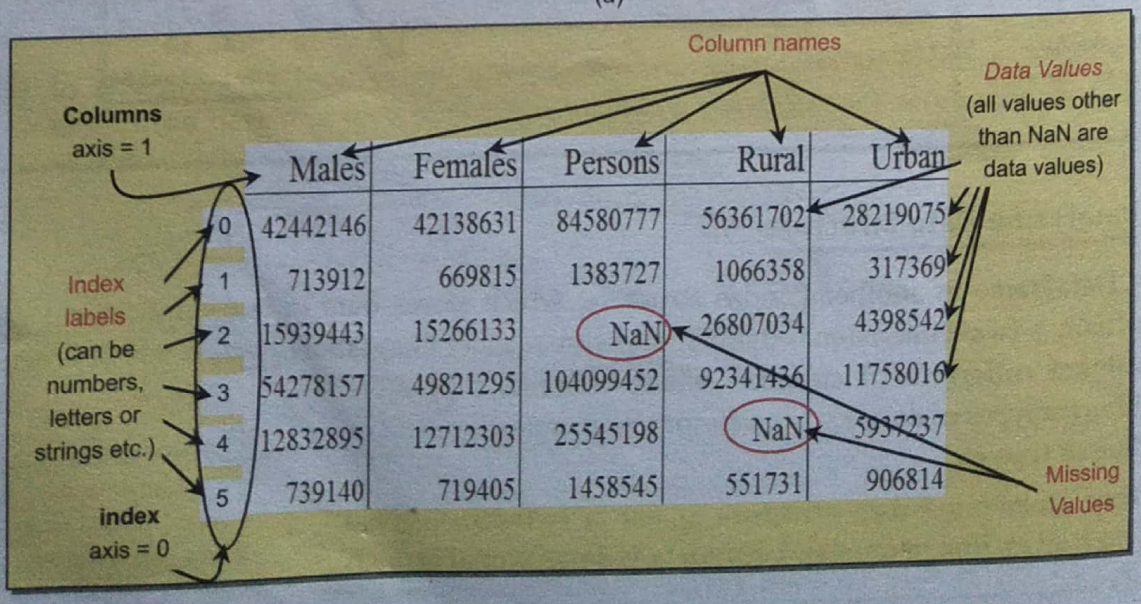
To see DataFrame Anatomy in action

Scan QR Code

	Statistic	*N*	*Mean*	*St. Dev.*	*Min*	*Max*
0	rating	30	64.633	12.173	40	85
1	complaints	30	66.600	13.315	37	90
2	privileges	30	53.133	12.235	30	83
3	learning	30	56.367	11.737	34	75
4	raises	30	64.633	10.397	43	88

	A	B	C	D	E	F
a	0.0	0.0	0.0	0.0	0.0	0.0
b	0.0	0.0	0.0	0.0	0.0	0.0
c	0.0	0.0	0.0	0.0	0.0	0.0
d	0.0	0.0	0.0	0.0	0.0	0.0
e	0.0	0.0	0.0	0.0	0.0	0.0

(*a*)



(*b*)

Figure 1.7 (*a*) Some sample DataFrame objects (*b*) Anatomy of a DataFrame object.

(*iv*) There is no condition of having all data of same type across columns; its columns can have data of different types. [Fig. 1.7(*a*)]

(*v*) You can easily change its values, *i.e.*, it is **value-mutable**.

(*vi*) You can add or delete rows/columns in a DataFrame. In other words, it is **size-mutable**.

Now that you have an idea about DataFrames, you can understand the terms associated with DataFrame as explained in following figure [Fig. 1.7(*b*)].

NOTE

DataFrames are both, **value-mutable** and **size-mutable**, *i.e.*, you can change both its values and size.

1.6 Creating and Displaying a DataFrame

A DataFrame object can be created by passing data in two-dimensional format. Like earlier, before you do anything with pandas module, make sure to import Pandas and NumPy modules, *i.e.*, give the following two import statements on your code or Python console :

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

To create a DataFrame object, you can use syntax as :

*Both **D** and **F** are capital letters*

```
<datFrameObject> = panda.DataFrame(<a 2D datastructure>,\
[columns = <column sequence> ], [index = <index sequence>])
```

Command continuation mark

where the 2D data structure passed to it, contains the data values. If you have imported Pandas as **pd**, then you can create a DataFrame as **pd.DataFrame( )** also.

You can create a DataFrame object by passing data in many different ways, such as :

(*i*) Two-dimensional dictionaries *i.e.*, dictionaries having lists or dictionaries or *ndarrays* or Series objects etc.

(*ii*) Two-dimensional *ndarrays* (NumPy array)

(*iii*) Series type object

(*iv*) Another DataFrame object

1. Creating a DataFrame Object from a 2-D Dictionary

A two dimensional dictionary is a dictionary having items as (key : value) where value part is a data structure of any type : another dictionary, an ndarray, a Series object, a list etc. But here the value parts of all the keys should have similar structure and equal lengths.

Following sections explain these with the help of many examples.

(*a*) Creating a dataframe from a 2D dictionary having values as lists/ndarrays :

You can have a two-dimensional dictionary wherein the value part consists of either *lists* or *ndarrays*. Passing such a 2D dictionary to *DataFrame( )* will create a dataframe object as you can see yourself in the example below :

```
>>> dict1 = {'Students' : ['Ruchika', 'Neha', 'Mark', 'Gurjyot', 'Jamaal'],
'Marks' : [ 79.5, 83.75, 74, 88.5, 89 ],
'Sport' : ['Cricket', 'Badminton', 'Football', 'Athletics', 'Kabaddi'] }
>>> dict1
{'Marks': [79.5, 83.75, 74, 88.5, 89],
'Sport': ['Cricket', 'Badminton', 'Football', 'Athletics', 'Kabaddi'],
'Students': ['Ruchika', 'Neha', 'Mark', 'Gurjyot', 'Jamaal']}
>>> dtf1 = pd.DataFrame(dict1)
>>> dtf1
```

	Marks	Sport	Students
0	79.50	Cricket	Ruchika
1	83.75	Badminton	Neha
2	74.00	Football	Mark
3	88.50	Athletics	Gurjyot
4	89.00	Kabaddi	Jamaal

The created DataFrame object has its index assigned automatically (0 onwards), and the columns created from keys, are placed in sorted order.

indexes are automatically generated

As you can see that the created DataFrame object has its index assigned automatically (0 onwards) just as it happens with Series objects, and the **columns are placed in sorted order**.

You can specify your own indexes too by specifying a sequence by the name **index** in the DataFrame( ) function, *e.g.*,

```
>>> dtf2 = pd.DataFrame(dict1, index = ['I', 'II', 'III', 'IV', 'V'])
>>> dtf2
```

*Specify the new indexes by giving **index sequence***

	Marks	Sport	Students
I	79.50	Cricket	Ruchika
II	83.75	Badminton	Neha
III	74.00	Football	Mark
IV	88.50	Athletics	Gurjyot
V	89.00	Kabaddi	Jamaal

See, the indexes are now as per the index sequence

If you specify the *index sequence*, Python will take indexes from the **index** sequence, BUT the number of indexes given in the index sequence MUST MATCH the length of dictionary's values, otherwise Python will give error (*see* below) :

```
In [32]: dtf2 = pd.DataFrame(dict1, index =['I', 'II', 'III', 'IV', 'V', 'VI'])
Traceback (most recent call last):
      .
      .
      .
  File "C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\internals.py", line
4608, in construction_error
    passed, implied))

ValueError: Shape of passed values is (3, 5), indices imply (3, 6)
```

EXAMPLE 21 *Given a dictionary that stores the section names' list as value for 'Section' key and contribution amounts' list as value for 'Contri' key :*

```
dict1 = {'Section':[ 'A', 'B', 'C', 'D'],
         'Contri': [ 6700, 5600, 5000, 5200] }
```

Write code to create and display the data frame using above dictionary.

SOLUTION

```
import pandas as pd
dict1 = {'Section':[ 'A', 'B', 'C', 'D'],
         'Contri':[ 6700, 5600, 5000, 5200] }
df1 = pd.DataFrame(dict1)
print(df1)
```

Output

	Contri	Section
0	6700	A
1	5600	B
2	5000	C
3	5200	D

(b) Creating a dataframe from a 2D dictionary having values as dictionary objects :

A 2D dictionary can have values as dictionary objects too. You can also create a data frame object using such 2D dictionary object, (*e.g., see* below) :

Given a 2D dictionary namely **people** :

```
people = {'Sales' : {'name': 'Rohit', 'age': '24', 'sex': 'Male' },
    'Marketing' : {'name': 'Neha', 'age': '25', 'sex': 'Female'}}
```

Here, the keys of inner dictionaries are exactly the same (*'name'*, *'age'*, and *'sex'*).

You can create a DataFrame by passing this dictionary :

```
>>> pd.DataFrame(people)
```

Inner dictionary keys as indexes

Out dictionary keys as columns

	Marketing	Sales
age	25	24
name	Neha	Rohit
sex	Female	Male

This time the keys of inner dictionaries make the indexes and the keys of outer dictionary make the columns.

EXAMPLE 22 *Create and display a DataFrame from a 2D dictionary,* ***Sales****, which stores the quarter-wise sales as inner dictionary for two years, as shown below :*

```
Sales = {'yr1' : { 'Qtr1' : 34500, 'Qtr2' : 56000, 'Qtr3' : 47000, 'Qtr4' : 49000 } ,
         'yr2' : { 'Qtr1' : 44900, 'Qtr2' : 46100, 'Qtr3' : 57000, 'Qtr4' : 59000 } }
```

SOLUTION

```
import pandas as pd
Sales = { 'yr1' : { 'Qtr1' : 34500, 'Qtr2' : 56000, 'Qtr3' : 47000, 'Qtr4' : 49000 },
          'yr2' : { 'Qtr1' : 44900, 'Qtr2' : 46100, 'Qtr3' : 57000, 'Qtr4' : 59000 } }
dfsales = pd.DataFrame(Sales)
print(dfsales)
```

Output

	yr1	yr2
Qtr1	34500	44900
Qtr2	56000	46100
Qtr3	47000	57000
Qtr4	49000	59000

In above example notice one thing : as the keys of all inner dictionaries (*yr1, yr2*) are exactly the same in number and names, the dataframe object **dfsales** also has the same number of indexes.

Now, had there been a situation **where inner dictionaries had non-matching keys**, then in that case Python would have done following things :

(*i*) There would have been **total number of indexes equal to sum of unique inner keys** in all the inner dictionaries.

(*ii*) For a key that has no matching keys in other inner dictionaries , value **NaN** would be used to depict the missing values.

```
>>> Collect1 = { 'yr1' :1500, 'yr2':2500}
>>> Collect2 = {'yr1':2200, 'Nil': 0}
```

Two dictionaries with dissimilar keys

```
>>> Collect = { 'I': Collect1, 'II':Collect2}
```

Dictionaries with dissimilar keys as inner dictionaries of a 2D dictionary

```
>>> df = pd.DataFrame(Collect)
```

DataFrame created with non-matching inner key dictionaries

```
>>> df
        I       II
Nil   NaN      0.0
yr1  1500.0  2200.0
yr2  2500.0     NaN
```

All the inner keys become indexes

NaN values added for non-matching keys of inner dictionaries

EXAMPLE 23 *Carefully read the following code.*

```
import pandas as pd
yr1 = { 'Qtr1' : 44900, 'Qtr2' : 46100, 'Q3' : 57000, 'Q4' : 59000 }
yr2 = { 'A' : 54500, 'B' : 51000, 'Qtr4' : 57000 }
diSales1 = { 1 : yr1 , 2 : yr2}
df3 = pd.DataFrame(diSales1)
```

(*i*) *list the index labels of the DataFrame df3*

(*ii*) *list the column names of DataFrame df3.*

SOLUTION

(*i*) The index labels of *df3* will include : A, B, Q3, Q4, Qtr1, Qtr2, Qtr4.
The total number of indexes is equal to total unique inner keys, *i.e.*, 7.

(*ii*) The column names of *df3* will be : 1, 2

NOTE

Total number of indexes in a DataFrame object are equal to total unique inner keys of the 2D dictionary passed to it and it would use NaN values to fill missing data *i.e.*, where the corresponding values for a key are missing in any inner dictionary.

2. Creating a DataFrame Object from a List of Dictionaries/Lists

If you pass a 2D list having dictionaries as its elements (**list of dictionaries**) to **pandas.DataFrame( )** function, it will create a DataFrame object such that the inner dictionary keys will become the columns and inner dictionary's values will make rows. *For example,*

```
>>> import pandas as pd
>>> topperA = { 'Rollno':115, 'Name': 'Pavni', 'Marks': 97.5}
>>> topperB = { 'Rollno':236, 'Name': 'Rishi', 'Marks': 98}
>>> topperC = { 'Rollno':307, 'Name' : 'Preet', 'Marks' : 98.5}
>>> topperD = { 'Rollno':422, 'Name': 'Paula', 'Marks': 98}
>>> toppers = [ topperA, topperB, topperC, topperD]
>>> topDf = pd.DataFrame(toppers)
>>> topDf
```

*This is a **list of dictionaries***

	Rollno	Name	Marks
0	115	Pavni	97.5
1	236	Rishi	98.0
2	307	Preet	98.5
3	422	Paula	98.0

See, the keys of inner dictionaries have made the columns of the DataFrame.

See, the values of inner dictionaries have made the rows of the DataFrame.

When a DataFrame is created with a list of dictionaries, the columns are created from the keys of dictionary elements and default indexes are assigned to rows (0, 1, 2, 3 ….) as you can see above. You can specify your own index labels through the **index** argument containing the row/index labels (see below).

```
>>> topDf = pd.DataFrame(toppers, index = ["Sec A","Sec B","Sec C","Sec D"] )
>>> topDf
```

	Rollno	Name	Marks
Sec A	115	Pavni	97.5
Sec B	236	Rishi	98.0
Sec C	307	Preet	98.5
Sec D	422	Paula	98.0

See, the row labels are created from the sequence given with index argument of DataFrame() function.

EXAMPLE 24 *Write a program to create a dataframe from a list containing dictionaries of the sales performance of four zonal offices. Zone names should be the row labels.*

SOLUTION

```
import pandas as pd
zoneA = {'Target':56000, 'Sales':58000}
zoneB = {'Target':70000, 'Sales':68000}
zoneC = {'Target':75000, 'Sales':78000}
zoneD = {'Target':60000, 'Sales':61000}
sales = [zoneA, zoneB, zoneC, zoneD]
saleDf = pd.DataFrame(sales , index = ['zoneA', 'zoneB', 'zoneC', 'zoneD'])
print( saleDf )
```

Output

	Target	Sales
zoneA	56000	58000
zoneB	70000	68000
zoneC	75000	78000
zoneD	60000	61000

Passing a 2D list, *i.e.*, a **list having lists** will also create a dataframe where each inner list will form the row of the dataframe, *e.g.*,

```
>>> list2 = [ [ 25, 45, 60], [34, 67, 89], [88, 90, 56] ]

>>> df1 = pd.DataFrame( list2 )
>>> df1
    0   1   2
0   25  45  60
1   34  67  89
2   88  90  56
```

See, the inner lists have formed a row each, in the created dataframe.

With 2D lists, the created dataframe by default names the columns and indexes as 0, 1, 2... unless you specify the index and columns arguments. You can specify own index names/labels with **index** argument and column labels with **columns** argument in the DataFrame( ) as illustrated in following two examples.

EXAMPLE 25 *Write a program to create a dataframe from a 2D list. Specify own index labels.*

SOLUTION

```
import pandas as pd
list2 = [ [ 25, 45, 60], [34, 67, 89], [88, 90, 56] ]
df2 = pd.DataFrame(list2, index = ['row1', 'row2', 'row3'] )
print ( df2 )
```

Output

	0	1	2
row1	25	45	60
row2	34	67	89
row3	88	90	56

EXAMPLE 26 *Write a program to create a dataframe from a list containing 2 lists, each containing Target and actual Sales figures of four zonal offices. Give appropriate row labels.*

SOLUTION

```
import pandas as pd
Target = [56000, 70000, 75000, 60000 ]
Sales = [58000, 68000, 78000, 61000]
ZoneSales = [Target, Sales]
zsaleDf = pd.DataFrame(ZoneSales, columns = ['ZoneA','ZoneB','ZoneC','ZoneD'],
                                                    index = ['Target', 'Sales'])

print( zsaleDf )
```

Output

	ZoneA	ZoneB	ZoneC	ZoneD
Target	56000	70000	75000	60000
Sales	58000	68000	78000	61000

3. Creating a DataFrame Object from a 2-D ndarray

You can also pass a two-dimensional NumPy array (*i.e.*, having ***shape*** as (<*n*>, <*n*>)) to DataFrame( ) to create a dataframe object.

Let us see how :

```
>>> narr1 = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> narr1.shape
(2, 3)
>>> dtf1 = pd.DataFrame(narr1)
>>> dtf1
```

	0	1	2
0	1	2	3
1	4	5	6

columns (header row) *indexes* (index column)

As you can see in above code example that Python created **indexes** or **axis = 0** from the first dimension of the passed ndarray and **columns (axis = 1)** from the second dimension of the passed ndarray. As no **keys** are there, hence default names are given to indexes and columns, *i.e.*, 0 onwards.

You can however, specify your own column names and /or index names by giving a **columns** sequence and/or index sequence :

```
>>> dtf2 = pd.DataFrame(narr1, columns = ['One', 'Two', 'Three'])
>>>dtf2
```

By giving a **columns** sequence, you can specify your own column names.

	One	Two	Three
0	1	2	3
1	4	5	6

See, this time the columns have the names given in columns sequence

```
>>> narr2 = np.array([[11.5, 21.2, 33.8], [40, 50, 60], [212.3, 301.5, 405.2]] )
>>> dtf3 = pd.DataFrame(narr2, columns = ['First', 'Second', 'Third'], index = ['A', 'B', 'C'])
>>> dtf3
```

By giving an **index** sequence, you can specify your own index names or labels .

You can specify either columns or index or both the sequences

	First	Second	Third
A	11.5	21.2	33.8
B	40.0	50.0	60.0
C	212.3	301.5	405.2

This time the columns and indexes have names or labels as per the given **columns** and **index** sequences respectively.

In above examples, the ndarrays that are passed to DataFrame have same number of elements in each of the rows. If, however, the rows of ndarrays differ in length, *i.e.*, if number of elements

in each row differ, then Python will create just single column in the dataframe object and the type of the column will be considered as **object** (see below).

```
>>> narr3 = np.array([[101.5, 201.2], [400, 50, 600, 700], [212.3, 301.5, 405.2]] )
>>> narr3
array([list([101.5, 201.2]), list([400, 50, 600, 700]),
       list([212.3, 301.5, 405.2])], dtype = object)
```

2 values, 4 values, 3 values

See the datatype of the ndarray is **object** this time

```
>>> dtf4 = pd.DataFrame(narr3)
>>> dtf4
                0
0       [101.5, 201.2]
1    [400, 50, 600, 700]
2  [212.3, 301.5, 405.2]
```

Single column created this time because the lengths of rows of ndarray did not match

EXAMPLE 27 *What will be the output of following code ?*

```
import pandas as pd
import numpy as np
arr1 = np.array([ [11, 12], [13, 14 ], [15, 16] ], np.int32)
dtf2 = pd.DataFrame(arr1)
print(dtf2)
```

SOLUTION

```
    0   1
0  11  12
1  13  14
2  15  16
```

EXAMPLE 28 *Write a program to create a DataFrame from a 2D array as shown below :*

101	113	124
130	140	200
115	216	217

Output

```
     0    1    2
0  101  113  124
1  130  140  200
2  115  216  217
```

SOLUTION

```
import pandas as pd
import numpy as np
arr2 = np.array([ [101, 113, 124], [130, 140, 200 ], [115, 216, 217] ] )
dtf3 = pd.DataFrame(arr2)
print(dtf3)
```

4. Creating a DataFrame Object from a 2D Dictionary with Values as Series Objects

You can also create a DataFrame object by using multiple Series objects. In a 2D dictionary, you can have the values parts as Series objects and then you can pass this dictionary as argument to create a DataFrame object.

We have following Series objects :

```
>>> staff = pd.Series([20, 36, 44])
>>> salaries = pd.Series([166000, 246000, 563000])
```

And a dictionary that stores these Series objects as values :

```
>>> school = {'people':staff, 'Amount':salaries }
```

You can create a DataFrame object from above 2D dictionary by passing its name as argument :

```
>>> dtf4 = pd.DataFrame(school)
```

Now the DataFrame **dtf4**, created above, will be like :

```
>>> dtf4
   Amount      people
0  166000        20
1  246000        36
2  563000        44
```

EXAMPLE 29 *Consider two series objects* ***staff*** *and* ***salaries*** *that store the number of people in various office branches and salaries distributed in these branches, respectively.*

Write a program to create another Series object that stores average salary per branch and then create a DataFrame object from these Series objects.

SOLUTION

```
import pandas as pd
import numpy as np
staff= pd.Series([20, 36, 44])
salaries = pd.Series([279000, 396800, 563000])
avg = salaries / staff          # it will create avg series object
org= {'people':staff, 'Amount':salaries, 'Average':avg }
dtf5 = pd.DataFrame(org)
print(dtf5)
```

Output

```
   Amount      Average          people
0  279000  13950.000000         20
1  396800  11022.222222         36
2  563000  12795.454545         44
```

---

5. Creating a DataFrame Object from another DataFrame Object

You can pass an existing DataFrame object to **DataFrame( )** and it will create another dataframe object having similar data. Consider the code shown below that passes to **DataFrame( )** another dataframe object.

Given a DataFrame object **dtf1** :

```
>>> dtf1
   0  1  2
0  1  2  3
1  4  5  6
```

You can create an identical dataframe by passing its name (dtf1) to DataFrame( ) :

```
dfnew = pd.DataFrame(dtf1)
```

You can display the new DataFrame object to confirm that it is identical to dtf1:

```
>>> dfnew
   0  1  2
0  1  2  3
1  4  5  6
```

Please note there are some other methods too for creating dataframe objects but covering those will be beyond the scope of the book.

Displaying a DataFrame

Displaying a dataframe is the same as the way you display other variables and objects, *i.e.*, on the console prompt, either type its name or give **print( )** command with the *dataframe* object as you have been doing till now.

NOTE

DataFrames can also be created from text/CSV files, which we shall cover in **Chapter 4 — Data Transfer between DataFrames and Flat files/MySQL**.

1.7 DataFrame Attributes

To see DataFrame Attributes *in action*

Scan QR Code

When you create a DataFrame object, all information related to it (such as its *size*, its *datatype* etc.) is available through its *attributes*. You can use these attributes in the following format to get information about the dataframe object.

```
<DataFrame object>.<attribute name>
```

Some common attributes of DatFrame object are listed in table below. The code examples for the usage of these attributes follow the Table 1.3

Table 1.3 *Common attributes of DataFrame Objects*

Attribute	Description
index	The index (row labels) of the DataFrame.
columns	The column labels of the DataFrame.
axes	Return a list representing both the axes (axis 0 *i.e.*, index and axis 1, *i.e.*, columns) of the DataFrame.
dtypes	Return the dtypes of data in the DataFrame.
size	Return an int representing the number of elements in this object.
shape	Return a tuple representing the dimensionality of the DataFrame.
values	Return a Numpy representation of the DataFrame.
empty	Indicator whether DataFrame is empty.
ndim	Return an int representing the number of axes/array dimensions.
T	Transpose index and columns.

We are using the following DataFrame (**dfn**) to display various attributes, counting, transpose etc.

```
>>> dfn
     Marketing     Sales
age      25          24
name     Neha        Rohit
sex      Female      Male
```

(a) Retrieving Various Properties of a DataFrame Object

To view the value of an attribute, just give its name with the dataframe's name as depicted in following examples :

```
>>> dfn.index
Index(['age', 'name', 'sex'], dtype = 'object')
>>> dfn.columns
Index(['Marketing', 'Sales'], dtype = 'object')
>>> dfn.axes
[Index(['age', 'name', 'sex'], dtype = 'object'),
Index(['Marketing', 'Sales'], dtype = 'object')]
>>> dfn.dtypes
Marketing    object
Sales        object
dtype:       object
```

```
>>> dfn.size
6
>>> dfn.shape
(3, 2)
>>> dfn.ndim
2
>>> dfn.empty
False
```

(b) Getting Number of Rows in a DataFrame

The **len(<DF object>)** will return the number of rows in a dataframe *e.g.*,

```
>>> len(dfn)
    3
```

(c) Getting Count of non-NA Values in DataFrame

Like Series, you can use **count( )** with dataframe too to get the count of non-NaN or non-NA values, but **count( )** with dataframe is little elaborate :

(*i*) If you do not pass any argument or pass 0 (default is 0 only), then it returns count of non-NA values for each column, *e.g.*,

```
>>> dfn.count()
Marketing  3
Sales      3
dtype: int64
```

dfn.count( ) or dfn.count(0) or dfn.count(axis = 'index') will produce the same result

```
>>> dfn.count(axis = 'index')
Marketing  3
Sales      3
dtype: int64
```

You may also pass argument **axis = 'index'** to get the same result as above.

(*ii*) If you pass argument as 1, then it returns count of non-NA values for each row, *e.g.*,

```
>>> dfn.count(1)
age   2
name  2
sex   2
dtype: int64
```

dfn.count(1) or dfn.count(axis='columns') will produce the same result

```
>>> dfn.count(axis = 'columns')
age   2
name  2
sex   2
dtype: int64
```

You may also pass argument **axis = 'columns'** to get the same result as above.

(d) Transposing a DataFrame

You can transpose a dataframe by swapping its indexes and columns by using attribute T as shown below :

```
>>> dfn.T
          age  name   sex
Marketing  25  Neha   Female
Sales      24  Rohit  Male
```

Compare the transpose(dfn.T) with original **dfn** :

```
>>> dfn
      Marketing  Sales
age   25         24
name  Neha       Rohit
sex   Female     Male
```

EXAMPLE 30 *Write a program to create a DataFrame to store weight, age and names of 3 people. Print the DataFrame and its transpose.*

SOLUTION

```
import pandas as pd
# Creating the DataFrame
df = pd.DataFrame({'Weight' :[42, 75, 66],
                   'Name':['Arnav', 'Charles', 'Guru'],
                   'Age' :[15, 22, 35]})
print('Original Dataframe')
print(df)
print('Transpose:')
print(df.T)
```

Output

```
Original Dataframe
   Age     Name  Weight
0   15    Arnav      42
1   22  Charles      75
2   35     Guru      66
Transpose:
              0        1     2
Age          15       22    35
Name      Arnav  Charles  Guru
Weight       42       75    66
```

NOTE

You can also use **shape[0]** to see the number of rows and **shape[1]** for getting number of columns, *i.e.*,

```
df.shape[0]
df.shape[1]
```

(e) Numpy Representation of DataFrame

You can represent the values of a dataframe object in numpy way using *values* attribute :

```
>>> dfn.values
array([['25', '24'],
       ['Neha', 'Rohit'],
       ['Female', 'Male']], dtype = object)
```

1.8 Selecting or Accessing Data

From a DataFrame object, you can extract or select desired rows and columns as per your requirement. Let us see how.

For all the examples in this section, in coming lines, we are using the following DataFrame **dtf5** :

DataFrame : dtf5

	Population	Hospitals	Schools
Delhi	10927986	189	7916
Mumbai	12691836	208	8508
Kolkata	4631392	149	7226
Chennai	4328063	157	7617

1.8.1 Selecting/Accessing a Column

Selecting a column is easy, just use the following syntax :

```
<DataFrame object> [ <column name>]
```
Using square brackets

Or

```
<DataFrame object>.<column name>
```
Using dot notation

Now, consider the following example accessing columns ***Population, Schools*** from dataframe **dtf5**.

```
>>> dtf5['Population']
Delhi       10927986
Mumbai      12691836
Kolkata     4631392
Chennai     4328063
Name: Population, dtype: int64
```

```
>>> dtf5['Schools']
Delhi       7916
Mumbai      8508
Kolkata     7226
Chennai     7617
Name : Schools, dtype : int64
```

In the dot notation, make sure not to put any quotation marks around the column name.

For example,

```
>>> dtf5.Population
Delhi       10927986
Mumbai      12691836
Kolkata     4631392
Chennai     4328063
Name: Population, dtype: int64
```

While using dot nation, do not put any quotes around the column name

1.8.2 Selecting/Accessing Multiple Columns

To select multiple columns, you can give a list having multiple column names inside the square brackets with dataframe object, *i.e.*, as follows :

```
<DataFrame object> [[<column name>, <column name>, <column name>, ...]]
```

For example,

Notice double square brackets

```
>>> dtf5[ ['Schools', 'Hospitals'] ]
          Schools     Hospitals
Delhi       7916          189
Mumbai      8508          208
Kolkata     7226          149
Chennai     7617          157
```

List having multiple column names given inside the square brackets

Columns appear in the order of column names given in the **list inside square brackets** (see below). Compare it with above result too.

```
>>> dtf5[ ['Hospitals', 'Schools'] ]
          Hospitals     Schools
Delhi       189          7916
Mumbai      208          8508
Kolkata     149          7226
Chennai     157          7617
```

Columns appear in the order of column names given in the list inside the square brackets

EXAMPLE 31 *Given a DataFrame namely* ***aid*** *that stores the aid by NGOs for different states :*

	Toys	Books	Uniform	Shoes
Andhra	7916	6189	610	8810
Odisha	8508	8208	508	6798
M.P.	7226	6149	611	9611
U.P.	7617	6157	457	6457

Write a program to display the aid for

(i) Books and Uniform only *(ii) Shoes only*

SOLUTION

```
import pandas as pd
:          # DataFrame aid created or loaded
print("Aid for books and uniform:")
print(aid[['Books', 'Uniform'] ])
print("Aid for shoes:")
print(aid.Shoes)
```

Output

```
Aid for books and uniform:
         Books  Uniform
Andhra    6189      610
Odisha    8208      508
M.P.      6149      611
U.P.      6157      457
Aid for shoes:
Andhra    8810
Odisha    6798
M.P.      9611
U.P.      6457
Name: Shoes, dtype: int64
```

1.8.3 Selecting/Accessing a Subset from a DataFrame using Row/Column Names

To access row(s) and/or a combination of rows and columns, you can use following syntax to select/access a subset from a dataframe object :

```
<DataFrameObject>.loc [<startrow> : <endrow>, <startcolumn> : <endcolumn>]
```

The above syntax is a general syntax through which you can single/multiple rows /columns. Let us see some examples :

- **To access a row**, just give the row name/label as this : <DF object>.loc [<row label> , :]. Make sure not to miss the COLON AFTER COMMA.

```
>>> dtf5.loc['Delhi', :]
Population      10927986
Hospitals            189
Schools             7916
Name: Delhi
```

Use this syntax to access one row : <DF>.loc [<row label> , :]

```
>>> dtf5.loc['Chennai', :]
Population      4328063
Hospitals           157
Schools            7617
Name: Chennai
```

- **To access multiple rows**, use : <DF object>.loc [<start row > : <end row>, :]. Make sure not to miss the COLON AFTER COMMA.

```
>>> dtf5.loc['Mumbai' :'Kolkata', :]
           Population  Hospitals  Schools
Mumbai       12691836        208     8508
Kolkata       4631392        149     7226
```

Give start and end row indexes with <DFobject>.loc. Make sure not to forget comma and colon after comma

Please note that when you specify <start row> :<end row>, the Python will return all rows falling between ***start row*** and ***end row***, along with ***start row*** and ***endrow***. (see below)

```
>>> dtf5.loc['Mumbai' :'Chennai', :]
           Population  Hospitals  Schools
Mumbai       12691836        208     8508
Kolkata       4631392        149     7226
Chennai       4328063        157     7617
```

To see subset from a dataframe in action

Scan QR Code

- **To access selective columns**, use : <DF object>.loc [: , <start column > : <end column>]. Make sure not to miss the COLON BEFORE COMMA. Like rows, all columns falling between start and end columns, will also be listed :

```
>>> dtf5.loc[ : , 'Population' :'Schools']
           Population  Hospitals  Schools
Delhi        10927986        189     7916
Mumbai       12691836        208     8508
Kolkata       4631392        149     7226
Chennai       4328063        157     7617
```

All columns falling between start and end columns, are listed

```
>>> dtf5.loc[ : , 'Population' :'Hospitals']
           Population  Hospitals
Delhi        10927986        189
Mumbai       12691836        208
Kolkata       4631392        149
Chennai       4328063        157
```

To access range of columns from a range of rows, use :

```
<DF object>.loc [<startrow> : <endrow>, <startcolumn> : <endcolumn>].
```

```
>>>dtf5.loc['Delhi' : 'Mumbai' , 'Population' :'Hospitals']
          Population  Hospitals
Delhi     10927986    189
Mumbai    12691836    208
```

Selecting a range of columns from a range of rows

EXAMPLE 32 *Given a DataFrame namely* **aid** *that stores the aid by NGOs for different states :*

	Toys	Books	Uniform	Shoes
Andhra	7916	6189	610	8810
Odisha	8508	8208	508	6798
M.P.	7226	6149	611	9611
U.P.	7617	6157	457	6457

Write a program to display the aid for states 'Andhra' and 'Odisha' for Books and Uniform only.

SOLUTION

```
import pandas as pd
:       # DataFrame aid created or loaded
print( aid.loc['Andhra':'Odisha', 'Books':'Uniform'])
```

Output

	Books	Uniform
Andhra	6189	610
Odisha	8208	508

1.8.4 Selecting Rows/Columns from a DataFrame

Sometimes your dataframe object does not contain row or column labels or even you may not remember them. In such cases, you can extract subset from dataframe using the row and column ***numeric index/position***, but this time you will use **iloc** instead of **loc**. **iloc** means ***integer location*** .

```
<DF object>.iloc[<start row index> : <end row index>, <start col index> : <end column index>]
```

When you use **iloc**, then **<startindex> :<end index>** given for rows and columns work like slices, and the end index is excluded (unlike **loc**), just as in the slices.

Consider the following example :

```
>>> dtf5.iloc[0:2, 1:3]
        Hospitals   Schools
Delhi       189     7916
Mumbai      208     8508

>>> dtf5.iloc[0:2, 1:2]
        Hospitals
Delhi       189
Mumbai      208
```

With iloc, the end index is excluded in result

NOTE

With **loc**, both start label and end label are included when given as start : end, but with **iloc**, like slices end index/position is excluded when given as start : end.

The **loc** and **iloc** are very flexible and can be used in variety of ways as shown below.

EXAMPLE 33 *Consider a dataframe* **df** *as shown below.*

Sample DataFrame *df* (Reference 1.7*a*)

	Item Type	Sales Channel	Order Date	Order ID	Total Revenue	Total Cost	Total Profit
0	Cosmetics	Online	5/22/2017	8985	793518.00	477943.95	315574.05
1	Cereal	Online	5/20/2017	5559	1780539.20	1013704.16	766835.04
2	Personal Care	Offline	5/8/2017	4567	523807.57	363198.03	160609.54
3	Personal Care	Online	3/11/2017	6992	246415.95	170860.05	75555.90
4	Snacks	Online	2/25/2017	7562	1117953.66	713942.88	404010.78
5	Household	Offline	2/8/2017	52284	5997054.98	4509793.96	1487261.02
6	Meat	Online	1/14/2017	8253	2011149.63	1738477.23	272672.40
7	Clothes	Online	1/13/2017	1873	902980.64	296145.92	606834.72
8	Cosmetics	Online	12/31/2016	331438481	3876652.40	2334947.11	1541705.29
9	Office Supplies	Offline	12/6/2016	6213	617347.08	497662.08	119685.00
10	Cosmetics	Offline	11/19/2016	419123971	3039414.40	1830670.16	1208744.24
11	Cosmetics	Online	11/15/2016	286959302	2836990.80	1708748.37	1128242.43
12	Beverages	Offline	10/23/2016	345718562	221117.00	148141.40	72975.60
13	Clothes	Offline	7/25/2016	807025039	600821.44	197048.32	403773.12
14	Snacks	Online	6/30/2016	795490682	339490.50	216804.00	122686.50

Write statements to do the following :

(*i*) *Display rows 2 to 4 (both inclusive)*

(*ii*) *From rows 2 to 4 (both inclusive), display columns, 'Item Type' and 'Total Profit'.*

(*iii*) *From rows 2 to 4 (both inclusive), display first four columns.*

SOLUTION

(*i*) `>>> df [2:5]`

```
In [14]: df[ 2:5 ]
Out[14]:
      Item Type Sales Channel Order Date  Order ID  Total Revenue  Total Cost  Total Profit
2  Personal Care       Offline   5/8/2017      4567      523807.57   363198.03     160609.54
3  Personal Care        Online  3/11/2017      6992      246415.95   170860.05      75555.90
4         Snacks        Online  2/25/2017      7562     1117953.66   713942.88     404010.78
```

(*ii*) `>>> df.loc [ 2:5 , ['Item Type', 'Total Profit']]`

```
In [18]: df.loc [ 2:5 , ['Item Type', 'Total Profit'] ]
Out[18]:
       Item Type  Total Profit
2  Personal Care     160609.54
3  Personal Care      75555.90
4         Snacks     404010.78
5      Household    1487261.02
```

(*iii*) `>>> df.iloc[2:5,0:4]`

```
In [19]: df.iloc[2:5,0:4]
Out[19]:
      Item Type Sales Channel Order Date  Order ID
2  Personal Care       Offline   5/8/2017      4567
3  Personal Care        Online  3/11/2017      6992
4         Snacks        Online  2/25/2017      7562
```

Recall that when you use iloc, then <startindex> :<end index> given for rows and columns work like slices, and the end index is excluded

1.8.5 Selecting/Accessing Individual Value

To select/access an individual data value from a dataframe, you can use any of the following methods :

(*i*) Either give name of row or numeric index in square brackets with, *i.e.*, as this :

```
<DF object>.<column> [<row name or row numeric index>]
```

Consider the following examples :

```
>>> dtf5.Population['Delhi']
10927986
>>> dtf5.Population[1]
12691836
```

Either give name of row or numeric index with **DFobject.column[ ]** in square brackets. Both will yield same result

(*ii*) You can use **at** or **iat** attributes with DF object as shown below :

Use	To
`<DFobject>.at[<row label>, col label>]`	Access a single value for a row/ column label pair.
`<DFobject>.iat[<row index no.>, col index no.>,]`	Access a single value for a row/ column pair by integer position.

Consider examples given below :

```
>>> dtf5.at['Chennai', 'Schools']
7617
>>> dtf5.iat[3, 2]
7617
```

You can give *row and column names* with **at** attribute and *numeric row index and numeric column index* with **iat** attribute to access individual data value

1.9 Adding/Modifying Rows'/Columns' Values in DataFrames

You can assign or modify data in a dataframe in the same way as you do with other objects. All you need to do is to specify the row name and/or column name along with the dataframe's name. The process of adding and modifying rows'/columns' value is similar, as you will see in the following sub-sections.

1.9.1 Adding/Modifying a Column

Columns in a dataframe can be referred to in multiple ways. Assigning a value to a column :

- will modify it, if the column already exists.
- will add a new column, if it does not exist already.

To change or add a column, use syntax :

```
<DF object >.<column name> = <new value>
```

or

```
<DF object >.[<column>] = <new value>
```

If the given column name does not exist in dataframe then a new column with this name is added :

```
>>> dtf5['Density'] = 1219
>>> dtf5
          Population  Hospitals  Schools  Density
Delhi       10927986        189     7916     1219
Mumbai      12691836        208     8508     1219
Kolkata      4631392        149     7226     1219
Chennai      4328063        157     7617     1219
```

Since there is no column with name 'Density', Python added a new column with this name which has given value for all its rows

Although the above method adds a column BUT here the catch is that all the rows of this new column have the same given value. If you want to add a proper new column that has different values for all its rows, then you can assign the data values for each row of the column in the form of a list, *i.e.*, as shown below :

NOTE

if you assign something to <DF>[<columnname>] or using **at** or **loc** and if column name does not exist, then a new column is created and that has the same value for all its rows.

```
>>> dtf5['Density'] = [ 1500, 1219, 1630, 1050, 1100]
>>> dtf5
          Population  Hospitals  Schools  Density
Delhi      10927986.0      189.0   7916.0   1500.0
Mumbai     12691836.0      208.0   8508.0   1219.0
Kolkata     4631392.0      149.0   7226.0   1630.0
Chennai     4328063.0      157.0   7617.0   1050.0
Banglore    5678097.0     1200.0   1200.0   1100.0
```

This time Python has added new column with different values for all its row as per the list of values

Same way, you can modify an existing column by assigning a new list of values to it. That is, for existing column, it will change the data values and for non-existing column, **it will add a new column**. There are some other ways of adding a column to a dataframe. These are :

```
<DF object>.at[ : , <column name> ] = <values for column>
```

Or

```
<DF object>.loc[ : , <column name> ] = <values for column>
```

Or

```
<DF object>.= <DF object>.assign(<column name> =<values for column>)
```

For example, given below are some statements that will add a new column if the mentioned column name does not exist in a dataframe :

```
dtf5.loc[:, "Density"] = [1500, 1219, 1630, 1050, 1100]
dtf5 = dtf4.assign( Density = [1500, 1219, 1630, 1050, 1100] )
```

NOTE

When you assign something to a column of dataframe, then for existing column, it will change the data values and for non-existing column, it will add a new column.

You just need to make sure in above examples that the sequence which contains values for the new column must have values equal to number of rows in the dataframe otherwise Python will give error (error : **ValueError**).

1.9.2 Adding/Modifying a Row

Like columns, you can change or add rows to a DataFrame using **at** or **loc** attributes as explained below :

To change or add a row, use syntax :

```
<DF object>.at[<row name>, : ] = <new value>
```

Or

```
<DF object>.loc[<row name>, : ] = <new value>
```

Likewise, if there is no row with such row label, then Python adds new row with this *row label* and assigns given values to all its columns :

```
>>> dtf5.at['Banglore', :] = 1200
>>> dtf5
```

	Population	Hospitals	Schools	Density
Delhi	10927986.0	189.0	7916.0	1219.0
Mumbai	12691836.0	208.0	8508.0	1219.0
Kolkata	4631392.0	149.0	7226.0	1219.0
Chennai	4328063.0	157.0	7617.0	1219.0
Banglore	1200.0	1200.0	1200.0	1200.0

Since there is no row with label 'Banglore', Python added new row with this label which has same given value for all its columns

As you can see in the above output that if a mentioned row label with **at** or **loc** attributes does not exist in the DataFrame, Python will create a new row for it and this is how a new row is added to a DataFrame. But there is a catch – if you specify only a single value, then all the values in the newly added row will have the same value as it did in the above output.

You can add a new row by specifying individual values for each column. For this, specify all the values of the new row to be added as a sequence such as a list etc., *e.g.*,

```
>>> dtf5.at['Bangalore', :] = [10002980, 171, 7311, 1200]
>>> dtf5
```

See, this time values are specified for all the columns in the form of a list

	Population	Hospitals	Schools	Density
Delhi	10927986.0	189.0	7916.0	1219.0
Mumbai	12691836.0	208.0	8508.0	1219.0
Kolkata	4631392.0	149.0	7226.0	1219.0
Chennai	4328063.0	157.0	7617.0	1219.0
Banglore	10002980.0	171.0	7311.0	1200.0

See, this time the newly added row contains the values given in the list

While adding a row this way, make sure that the sequence containing values for different columns has **values for all the columns**, otherwise Python will raise **ValueError** (*see* below).

If you try to add a row having 4 values to the above DataFrame dtf5 having 5 columns, Python will give you error :

```
>>> dtf5.loc['Mohali', :] = [452980, 71, 211]
:
ValueError: cannot copy sequence with size 3 to array axis with dimension 4
```

This statement is trying to insert a row having 3 values in the dataframe with four columns and this the error.

NOTE

You can use **at** or **loc** attributes of a DataFrame to add/modify a row, column or individual cell.

EXAMPLE 34 *Consider the following dataframe* ***saleDf*** :

	Target	Sales
zoneA	56000	58000
zoneB	70000	68000
zoneC	75000	78000
zoneD	60000	61000

Write a program to add a column namely **Orders** *having values 6000, 6700, 6200 and 6000 respectively for the zones A, B, C and D. The program should also add a new row for a new zone* **ZoneE**. *Add some dummy values in this row.*

SOLUTION

```
import pandas as pd
:    # saleDf created or loaded here
saleDf['Orders'] = [6000, 6700, 6200, 6000 ]
saleDf.loc['zoneE', :]= [ 50000, 45000, 5000]
print(saleDf)
```

Output

	Target	Sales	Orders
zoneA	56000.0	58000.0	6000.0
zoneB	70000.0	68000.0	6700.0
zoneC	75000.0	78000.0	6200.0
zoneD	60000.0	61000.0	6000.0
zoneE	50000.0	45000.0	5000.0

1.9.3 Modifying a Single Cell

You may use <DataFrame>.iat[<row position>, <column position>] to modify values using row and column position. (*Refer Multiple Choice Question 22 given in Objective type questions at the end of this chapter.*)

To change or modify a single data value, use syntax :

```
<DF>.<columnname>[<row name/label>] = <new/modified value>
```

```
>>> dtf5.Population['Banglore'] = 5678097
>>> dtf5
```

	Population	Hospitals	Schools	Density
Delhi	10927986.0	189.0	7916.0	1219.0
Mumbai	12691836.0	208.0	8508.0	1219.0
Kolkata	4631392.0	149.0	7226.0	1219.0
Chennai	4328063.0	157.0	7617.0	1219.0
Banglore	5678097.0	1200.0	1200.0	1200.0

See, this time only this cell got modified

1.10 Deleting/Renaming Columns/Rows

Python Pandas provides two ways to delete rows and columns — **del** statement and **drop( )** function. Pandas also provides **rename( )** function to rename rows and columns. In this section, we shall talk about how rows/columns can be deleted or renamed.

Let us now talk about how you can delete or rename columns and rows.

1.10.1 Deleting Rows/Columns in a DataFrame

To delete a column, you use **del** statement as this :

```
del <Df object>[<column name>]
```

For example,

```
>>> del dtf5['Density']
>>> dtf5
          Population    Hospitals   Schools
Delhi     10927986.0    189.0       7916.0
Mumbai    12691836.0    208.0       8508.0
Kolkata   4631392.0     149.0       7226.0
Chennai   4328063.0     157.0       7617.0
Banglore  5678097.0     1200.0      1200.0
```

Column 'Density' deleted

To delete rows from a dataframe, you can use **<DF>.drop(index or sequence of indexes)**, *e.g.*,

Both these commands will delete the rows with indexes 2, 4, 6, 8, 12 from dataframe **df** :

```
df.drop(range(2, 13, 2))
df.drop([2, 4, 6, 8, 12])
```

Argument to drop( ) should be either an index, or a sequence containing indexes

You can also give **axis = 1** along with indexes/labels then **drop( )** will drop the columns, *i.e.*, the following command will drop the mentioned columns from dataframe df :

```
df.drop(["Total Cost", "Order ID"], axis = 1)
```

This argument signifies to delete columns

EXAMPLE 35 *From the dtf5 used above, create another DataFrame and it must not contain the column 'Population' and the row Bangalore.*

SOLUTION

```
import pandas as pd
:          # DataFrame dtf5 created or loaded
dtf6 = pd.DataFrame(dtf5)
del dtf6['Population']
dtf6 = dtf6.drop(['Bangalore])
print(dtf6)
```

Output

```
          Hospitals   Schools
Delhi     189.0       7916.0
Mumbai    208.0       8508.0
Kolkata   149.0       7226.0
Chennai   157.0       7617.0
```

1.10.2 Renaming Rows/Columns

To change the name of any row/column individually, you can use the rename( ) function of DataFrame as per the syntax given below.

```
<DF>.rename( index = {<names dictionary>}, columns = {<names dictionary>}, inplace = False )
```

where,

- The **index** argument is for index names (row labels). If you want to rename rows only then specify only **index** argument.
- The **columns** argument is for the **columns** names. If you want to rename **columns** only then specify only **columns** argument.
- For both **index** and **columns** arguments, specify the **names-change dictionary** containing original names and the new names in a form like {**old name: new name**}
- Specify **inplace** argument as *True* if you want to rename the rows/columns in the same dataframe as if you skip this argument, then a new dataframe is created with new indexes/columns' names and original remains unchanged.

Let us now practically see how **rename( )** works.

Consider the dataframe **topDf** shown below :

	Rollno	Name	Marks
Sec A	115	Pavni	97.5
Sec B	236	Rishi	98.0
Sec C	307	Preet	98.5
Sec D	422	Paula	98.0

To change the row labels as *'A', 'B', 'C', 'D'*, you can write :

```
>>> topDf.rename( index = {'Sec A':'A', 'Sec B':'B', 'Sec C':'C', 'Sec D':'D'} )
```

	Rollno	Name	Marks
A	115	Pavni	97.5
B	236	Rishi	98.0
C	307	Preet	98.5
D	422	Paula	98.0

See, the names dictionary for index argument is storing the old and new index names

*The output of **rename()** has shown the changed indexes but are these changed in original **topDf** as well ?*

The above statement will show the changed indexes but when you display the dataframe **topDf** after executing above statement, it will show you the original dataframe (*see* below) because by default **rename( )** creates a new dataframe with changed names.

```
>>> topDf
```

	Rollno	Name	Marks
Sec A	115	Pavni	97.5
Sec B	236	Rishi	98.0
Sec C	307	Preet	98.5
Sec D	422	Paula	98.0

See, the rename( ) function by default does not make changes in the original dataframe; it creates a new dataframe with the changes and the original dataframe remains unchanged

*See the original dataframe **topDf** is unchanged*

To make changes in the index/columns' names original dataframe, you need to specify additional argument **inplace = True** in the **rename( )**, *e.g.,*

```
>>>topDf.rename(index = {'Sec A':'A', 'Sec B':'B', 'Sec C':'C', 'Sec D':'D'}, inplace = True)
>>> topDf
   Rollno  Name   Marks
A  115     Pavni  97.5
B  236     Rishi  98.0
C  307     Preet  98.5
D  422     Paula  98.0
```

By giving inplace =True argument, it will now make the changes in the original dataframe

*With **inplace** as True, **rename()** has changed the indexes in original **topDf***

You can use **rename( )** to change columns names too, e.g., following statement will change the columns names in the **topDf** dataframe. (Consider the original **topDf** shown earlier in the beginning of this section) You can choose to change selective columns' names or all the columns' names. Only the **names** given in the **name-change dictionary** will get changed. *For example,*

```
>>> topDf.rename(columns ={'Rollno':'Rno'})
       Rno  Name   Marks
Sec A  115  Pavni  97.5
Sec B  236  Rishi  98.0
Sec C  307  Preet  98.5
Sec D  422  Paula  98.0
```

the name-change dictionary containing old and new names.

You can combine any of these arguments together : **index, columns, inplace** arguments. Following examples are illustrating the same.

EXAMPLE 36 *Consider the **saleDf** shown below.*

	Target	Sales
zoneA	56000	58000
zoneB	70000	68000
zoneC	75000	78000
zoneD	60000	61000

Write a program to rename indexes of 'zoneC' and 'zoneD' as 'Central' and 'Dakshin' respectively and the column names 'Target and 'Sales' as 'Targeted' and 'Achieved' respectively.

SOLUTION

```
import pandas as pd
:  # saleDf created or loaded here
print(saleDf.rename(index = {'zoneC':'Central', 'zoneD':'Dakshin'}, \
   columns = {'Target':'Targeted', 'Sales':'Achieved'}) )
```

Output

	Targeted	Achieved
zoneA	56000	58000
zoneB	70000	68000
Central	75000	78000
Dakshin	60000	61000

EXAMPLE 37 *Will the previous program reflect the renamed indexes and columns' names in the dataframe saleDf ? Make changes in the previous program so that dataframe* ***saleDf*** *has the changed indexes and columns.*

SOLUTION

The previous program will not reflect the changed indexes and columns' names in the dataframe saleDf on which rename( ) is applied because **inplace = True** argument is missing. The changed program to make changes in saleDf would be :

```
import pandas as pd
:    # saleDf created or loaded here
saleDf.rename( index = {'zoneC':'Central', 'zoneD':'Dakshin'}, \
               columns = {'Target':'Targeted', 'Sales':'Achieved'}), inplace = True )
print(saleDf)
```

1.11 More on DataFrame Indexing — BOOLEAN INDEXING

Till now you have learnt to create and use DataFrames in variety of ways. You have also learnt to change *indexes, column names, rename* them etc. Let us now talk about an interesting feature of DataFrames – **Boolean Indexing**.

Boolean Indexing, as the name suggests, means having Boolean Values [(*True or False*) or (1 or 0) sometimes] as indexes of a dataframe. You might be wondering – Why? What is the need for having indexes as *True* or *False* ?

Well, your question is genuine and so is the answer. Actually, in some situations, we may need to divide our data in two subsets – *True* or *False, e.g.*, your school has decided to launch online classes for you. But some days of the week are designated for it. So, a dataframe related to this information might look like :

The Boolean indexes divide the dataframe in two groups – *True* rows and *False* rows.

	Day	No. of Classes
True	Monday	6
False	Tuesday	0
True	Wednesday	3
False	Thursday	0
True	Friday	8

The Boolean indexes divide the dataframe in *two* groups – *True rows* and *False rows*. This is useful division in situations where you find out things like – On which days, the *online classes* are held ? Or Which ones are *offline classes* days ? And so on.

Boolean indexes can either be in True/False form or in (1 or 0) form.

BOOLEAN INDEXING

> **Boolean Indexing** refers to having Boolean Values [(True or False) or (1 or 0)] as indexes of a dataframe.

Check Point 1.1

1. What is the use of Python Pandas library ?
2. Name the Pandas object that can store one dimensional array like object and can have numeric or labelled indexes.
3. Can you have duplicate indexes in a series object ?
4. What do these attributes of series signify ? (*a*) size (*ii*) itemsize (*iii*) nbytes
5. If S1 in a series object then how will *len(S1)* and *S1.count( )* behave ?
6. What are NaNs ? How do you store them in a datastructures ?
7. True/False. Series objects always have indexes 0 to $n-1$.
8. What is the use of del statement ?
9. What does drop( ) function do ?
10. What is the role of **inplace** argument in rename( ) function.

1.11.1 Creating Dataframes with Boolean Indexes

Let us first create a dataframe with Boolean indexes *True* and *False*. While creating a dataframe with Boolean indexes ***True*** and ***False***, make sure that *True* and *False* are not enclosed in quotes (*i.e.*, like 'true' or 'False'), otherwise it will give you error (**KeyError**) while accessing data with Boolean indexes using **.loc**, because 'True' and 'False' are string values, not Boolean values.

The ***solved problem 27*** depicts the same problem.

Let us first create above shown dataframe containing online classes' information, through the code given below.

```
import pandas as pd
Days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ]
Classes = [ 6, 0, 3, 0, 8]
dc = {'Days':Days, 'No. of Classes':Classes}
clasDf = pd.DataFrame(dc, index = [True, False, True, False, True] )
```

Boolean indexes provided for each row. Notice True and False values are Boolean values not strings (not enclosed in quotes

The Boolean values as indexes

```
>>> clasDf
       Days      No. of Classes
True   Monday    6
False  Tuesday   0
True   Wednesday 3
False  Thursday  0
True   Friday    8
```

You can also provide Boolean indexing to dataframes as ***1s*** and ***0s***. Let us create another dataframe (***clasDf1***) similar to the above shown dataframe (***clasDf***) having indexes as **1s** and **0s**, through the code given below.

```
import pandas as pd
Days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ]
Classes = [6, 0, 3, 0, 8]
dc = {'Days':Days, 'No. of Classes':Classes}
clasDf1 = pd.DataFrame(dc, index = [1, 0, 1, 0, 1] )
```

Boolean indexes provided for each row as 1s and 0s this time

```
>>> clasDf1
      Days      No. of Classes
1     Monday    6
0     Tuesday   0
1     Wednesday 3
0     Thursday  0
1     Friday    8
```

1.11.2 Accessing Rows from DataFrames with Boolean Indexes

Boolean indexing is very useful for filtering records, *i.e.*, for finding or extracting the *True* or *False* indexed rows. You can filter out records from dataframes with Boolean indexes using the **.loc** attribute as depicted below :

```
<DF>.loc[True]      # to display all records with True index
<DF>.loc[False]     # to display all records with False index
<DF>.loc[1]         # to display all records with index as 1
<DF>.loc[0]         # to display all records with index as 0
```

Use **.loc[ ]** with dataframes having Boolean indexes to filter out rows.

For example,

```
>>> clasDf.loc[True]
      Days        No. of Classes
True  Monday            6
True  Wednesday         3
True  Friday            8
```

See, the result now contains only the rows having index as True

```
>>> clasDf.loc[False]
      Days        No. of Classes
False Tuesday           0
False Thursday          0
```

See, the result now contains only the rows having index as False

```
>>> clasDf1.loc[0]
      Days        No. of Classes
0     Tuesday           0
0     Thursday          0
```

See, the result now contains only the rows having index as 0