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.

Scanned with CamScanner

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.

Scanned with CamScanner

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.

Scanned with CamScanner

(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)
```

Scanned with CamScanner

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>, ...]]
```

Scanned with CamScanner

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>]
```

Scanned with CamScanner

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
```

Scanned with CamScanner

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.

Scanned with CamScanner